TECHNISCHE UNIVERSITÄT DRESDEN

FAKULTÄT INFORMATIK
INSTITUT FÜR SOFTWARE- UND MULTIMEDIATECHNIK
PROFESSUR FÜR COMPUTERGRAPHIK UND VISUALISIERUNG
PROF. DR. STEFAN GUMHOLD

Großer Beleg

Anforderungsanalyse, Definition und Implementierung einer Netzwerkkommunikationsschicht für ein modulares Simulations- und VR-System

Markus Müller
(Mat.-Nr.: 2815730)

Betreuer: Dipl. Inf. (FH) Benjamin Neidhold

Dresden, 19. September 2006

Inhaltsverzeichnis

1 Einleitung **3**

2 Anforderungsanalyse **5**

2.1 Allgemeine Anforderungen an verteilte Virtual-Reality Umgebungen 6

2.2 Szenarioanalyse 7

2.2.1 Baggersimulator 7

2.2.2 Entwicklung neuer Komponenten 7

2.2.3 Kontrollzentrum 9

2.2.4 Verteilte Simulation 10

2.2.5 Multiplayer-Spiel 11

2.3 Abgeleitete Anforderungen an die Kommunikationsschicht 11

2.4 Programmiermodell und Aufrufsemantik 14

2.5 Evaluation der bestehenden Netzwerkkommunikationsschicht 16

3 Entwurf **19**

3.1 Sicht auf die Kommunikationsschicht 19

3.1.1 Unterstützte Funktionen und Datentypen 20

3.1.2 Client-Server-Architektur 22

3.1.3 Konfiguration 23

3.2 Schnittstelle zur Transportschicht 25

3.2.1 Unterstützte Transporttypen 27

3.2.2 Nachrichtenformat 29

3.3 Kommunikationssteuerung 30

4 Implementierung **33**

4.1 Überblick und Unterschiede der Implementierungen in C++ und C# 34

4.2 TCP 36

4.3 UDP 38

4.4 Broadcast 40

4.5 Kontrollzentrum 41
4.6 Automatische Generierung benutzerdefinierter Bibliotheken 42

5 Automatisiertes Testen 45

5.1 Arten von Tests 46
5.1.1 Unit Tests 46
5.1.2 Integrationstests 47
5.1.3 Systemtests 48
5.2 Testwerkzeuge 49
5.2.1 NUnit 50
5.2.2 Zanebug 53
5.2.3 CppUnit 55
5.2.4 TestDriven.NET 57
5.2.5 Visual Studio Team System 58
5.3 Testplanung 59
5.4 Testdurchführung 60
5.5 Testergebnisse 63

6 Ergebnisse 65

7 Ausblick 67

Literaturverzeichnis 69

A Listings 73

B Tabellen 89

1 Einleitung

Im Mai 2003 wurde am Institut für Fördertechnik, Baumaschinen und Logistik der TU Dresden ein interaktiver Forschungs- und Entwicklungssimulator für mobile Arbeitsmaschinen in Betrieb genommen. Gebaut wurde er von der EADS/Dornier GmbH Friedrichshafen in Zusammenarbeit mit Mitarbeitern der TU Dresden. Dabei entstand ein hochmodernes, modulares Simulations- und VR-System. Ein großer Teil der verwendeten Software war allerdings nicht quelltextoffen, was die Auswahl neuer Forschungs- und Anwendungsgebiete stark begrenzte.

Deshalb wurde 2005 von Joscha Metze das verteilte Kommunikations- und Visualisierungssystem ODO (Object Oriented Distributed Open Framework) vorgestellt [Met05]. Ziel der Arbeit war es, den Grundstein für ein flexibles VR-System zu legen das höchsten Erwartungen an verschiedenste virtuelle Welten gerecht wird. Der Lösungsansatz war, weitgehend unabhängige Komponenten zu schaffen, die Nachrichten über eine festgelegte Kommunikationsschnittstelle austauschen, so dass bestehende und neu entwickelte Komponenten unterschiedlicher Plattformen zu neuen Gesamtsystemen kombiniert werden können. Der Schwerpunkt lag, neben der Implementierung der Kommunikationsschicht, auch auf der Entwicklung einer von der eigentlichen Simulation entkoppelten, flexibel erweiterbaren Visualisierung, die höchsten qualitativen Ansprüchen gerecht wird.

Dieses Konzept wurde später zu einem verallgemeinerten VR-System erweitert [NMW05]. Durch den modularen Aufbau können verschiedene Komponenten wie Datenbasis, Visualisierung oder die eigentliche Simulation auf mehrere Rechner verteilt, neueste Technologien schnell integriert und heterogene Systeme vereint werden. Die eigens dafür entwickelte Netzwerkkommunikationsschicht bietet dem Anwendungsentwickler spezialisierte Schnittstellen, die er nur in einer XML-basierten IDL (Interface Definition Language) zu beschreiben braucht, um daraus mit einer XSL-Transformation automatisch den passenden C++-Code zu erzeugen.

Umfassende Ideen zur Erweiterung des Systems bedingen jedoch eine Evaluation und Erweiterung dieser Netzwerkkommunikationsschicht. Zusätzlich zur Plattformunabhängigkeit (Windows/Linux) soll durch die Integration von C# auch eine gewisse Unabhängigkeit von der Programmiersprache erreicht werden. Bei dieser Gelegenheit ist zu untersuchen, ob die momentane Definition der Schnittstelle alle aktuell benötigten Funktionen anbietet und inwieweit die bestehende Implementierung vereinfacht oder verbes-

sert werden kann. Um die Korrektheit der Implementierung weitgehend sicherzustellen, soll untersucht werden, welche Werkzeuge und Strategien für das automatisierte Testen der Netzwerkschicht geeignet sind.

2 Anforderungsanalyse

Das verteilte Kommunikations- und Visualisierungssystem ODO ist mittlerweile zu einem umfangreichen Framework für VR-Simulationen herangewachsen. Um die Entwicklung neuer Komponenten zu vereinfachen und damit zu beschleunigen, soll den am Projekt beteiligten Mitarbeitern und Studenten die Möglichkeit gegeben werden, die Programmiersprache C# und das Microsoft .NET-Framework zu nutzen.

C# ist einfacher zu beherrschen als C++, weniger fehleranfällig und durchgängig objektorientiert, was einen sauberen objektorientierten Entwurf erleichtert. Die umfangreiche Klassenbibliothek des .NET-Frameworks und das Microsoft Visual Studio .Net 2003/2005 verringern die Entwicklungszeiten neuer Anwendungen. Allerdings stellt die Entwicklungsumgebung für die Programmierung in C++ nicht dieselben Werkzeuge zur Verfügung wie für die Programmierung in C#. So werden z.B. für C++ noch nicht einmal grundlegendste Refactoringaufgaben wie das Umbenennen von Klassen oder Variablen unterstützt. Auch die automatische Anweisungsvervollständigung IntelliSense arbeitet beim Schreiben von C++-Code weniger komfortabel als beim Schreiben von C#-Code. Ein weiterer Nachteil ist, dass die automatische Generierung einer Dokumentation aus C++-Quellcode nicht von Visual Studio unterstützt wird, sondern externe Werkzeuge wie Doc++ oder Doxygen benötigt werden.

Um Komponenten, die in C# implementiert sind in das Gesamtsystem zu integrieren, ist eine Anpassung der Netzwerkkommunikationsschicht nötig. Prinizipiell könnte man dazu ohne allzu großen Aufwand eine Hüllklasse in C# schreiben, die den Zugriff auf die bestehende C++-Bibliothek erlaubt. Die Common Language Runtime von .NET bietet dazu einen Mechanismus namens „Platform Invoke“, der die Aufrufe von verwaltetem .NET-Code auf Aufrufe von nicht verwalteten Win32-Code abbildet [MSD06].

Auf der anderen Seite bietet eine Neuimplementierung die Chance, die bestehende Bibliothek noch einmal kritisch zu betrachten. Die Evolution des gesamten Systems könnte inzwischen neue Anforderungen an die Netzwerkkommunikationsschicht hervorgebracht haben, die nicht mehr optimal erfüllt werden. Zusätzlich zu den allgemeinen Anforderungen kann, weil die ursprüngliche Bibliothek schließlich schon von mehreren Entwicklern über einen längeren Zeitraum genutzt wird, auch die Erfüllung nicht explizit genannter Qualitätsmerkmale, vor allem die Benutzbarkeit, geprüft werden.

Deshalb, und weil sowohl die systematische Evaluation eines Softwaresystems, als auch die die Im-

plementierung begleitenden Tests, klar formulierte Anforderungen bedingen, soll trotz bestehender Implementierung zunächst ein Top-Down-Ansatz verfolgt werden. Dies beinhaltet die Analyse möglicher Szenarien und Anwendungsfälle, die Festlegung der benötigten Schnittstellen und die Auswertung der Erfahrungen der am Projekt beteiligten Personen.

2.1 Allgemeine Anforderungen an verteilte Virtual-Reality Umgebungen

Fünf zentrale Anforderungen an ein verteiltes Simulations- und VR-System wurden bereits in [Met05] herausgearbeitet und sollen hier noch einmal kurz zusammengefasst werden:

- Flexibilität: Der zentrale Aspekt im Design des Gesamtsystems ist die flexible Nutzbarkeit. Eine Vielzahl von neuen Anwendungsfeldern, Forschungsgebieten oder Technologien soll schnell und einfach erschließbar sein.
- Skalierbarkeit: Die Möglichkeiten der Plattform, auf der eine Anwendung läuft, sollen optimal genutzt werden. Zur Leistungssteigerung soll eine transparente Verteilung der Anwendung auf mehrere Rechner möglich sein.
- Heterogenität: Die Komponenten, und damit auch die Kommunikation zwischen ihnen, sollten auf unterschiedlichen Plattformen lauffähig sein.
- Synchronisation: Da alle Komponenten unabhängig von einander lauffähig sein sollten, kann es zur Konkurrenz zwischen ihnen kommen. Der Zugriff auf gemeinsame Daten muss synchronisiert werden.
- keine Diskontinuitäten: Zeitliche oder örtliche Diskontinuitäten führen zum Verlust der Immersion des Benutzers - er wird an die Existenz der Maschine erinnert.

Drei weitere wichtige Anforderungen aus [CDK01], die für verteilte Systeme jeder Art von Bedeutung sind, wurden nicht explizit genannt:

- Transparenz: Bestimmte Aspekte der Verteilung sollen dem Anwendungsentwickler verborgen bleiben.
- Fehlerbehandlung: Jeder Prozess, jeder Computer und jedes Netzwerk kann unabhängig von den anderen Teilen des Systems Fehler verursachen. Eine Komponente muss angemessen darauf reagieren können.
- Sicherheit: Verfügbarkeit, Vertraulichkeit und Integrität müssen gewährleistet sein.

Die Erfahrungen mit der bestehenden Implementierung haben gezeigt, dass eine weitere Eigenschaft nicht fehlen darf:

- Verwaltbarkeit: Konfigurationsmanagement, eine globale Sicht auf alle Komponenten, sowie eine zentrale Zustandskontrolle sind im alltäglichen Betrieb unerlässlich.

2.2 Szenarioanalyse

In einem flexibel gestalteten, verteilten Simulations- und VR-System sind zahlreiche Kombinationen der einzelnen Module denkbar. Um die daraus entstehenden Anforderungen an die Netzwerkkommunikationsschicht ableiten zu können, müssen möglichst vielfältige Szenarien betrachtet werden. Dabei ist es sicherlich hilfreich, von den konkreten, bereits bekannten Konfigurationen auszugehen, um diese dann um neue Systeme zu erweitern.

Zur Veranschaulichung sollen Diagramme verwendet werden, die auf UML-Verteilungs- und Komponentendiagrammen basieren. Die Quader beschreiben dabei einzelne Computer, welche immer einen Namen haben. Darin enthalten sind Komponenten eines bestimmten Typs, bei denen der Name optional ist. Die Pfeile stellen abstrakte, gerichtete Kommunikationsbeziehungen dar.

2.2.1 Baggersimulator

Die momentane Konfiguration des Baggersimulators ist aus Diagramm 2.1 ersichtlich.

Das Herz dieses Systems ist die Simulation, sie empfängt die Eingabedaten wie Beschleunigung oder Lenkrichtung von der Baggerkabine (CAB-Modul). Daraus, und mit Hilfe der Terraininformation aus der Datenbank, berechnet sie den neuen Systemzustand und sendet die resultierenden Parameter an die Ausgabemodule Motion, Sound und Visualisierung. Die Manipulation der Landschaft, also das Baggern, ist noch nicht implementiert. Die Visualisierung ist meist mehrfach instanziiert: drei Instanzen für die Projektion in der Kuppel, zwei Instanzen für die Rückspiegel und eine Instanz für den Kontrollraum. Bei Präsentationen vor größerem Publikum werden zwei weitere Instanzen für eine Powerwall benötigt. Alle anderen Module laufen auf jeweils genau einem Rechner.

2.2.2 Entwicklung neuer Komponenten

Die Entwicklung neuer Komponenten geschieht normalerweise zunächst auf einem einzelnen Rechner. Die Zusammenarbeit eines neuen Moduls, etwa einer neuen Simulation, mit den bestehenden Modulen

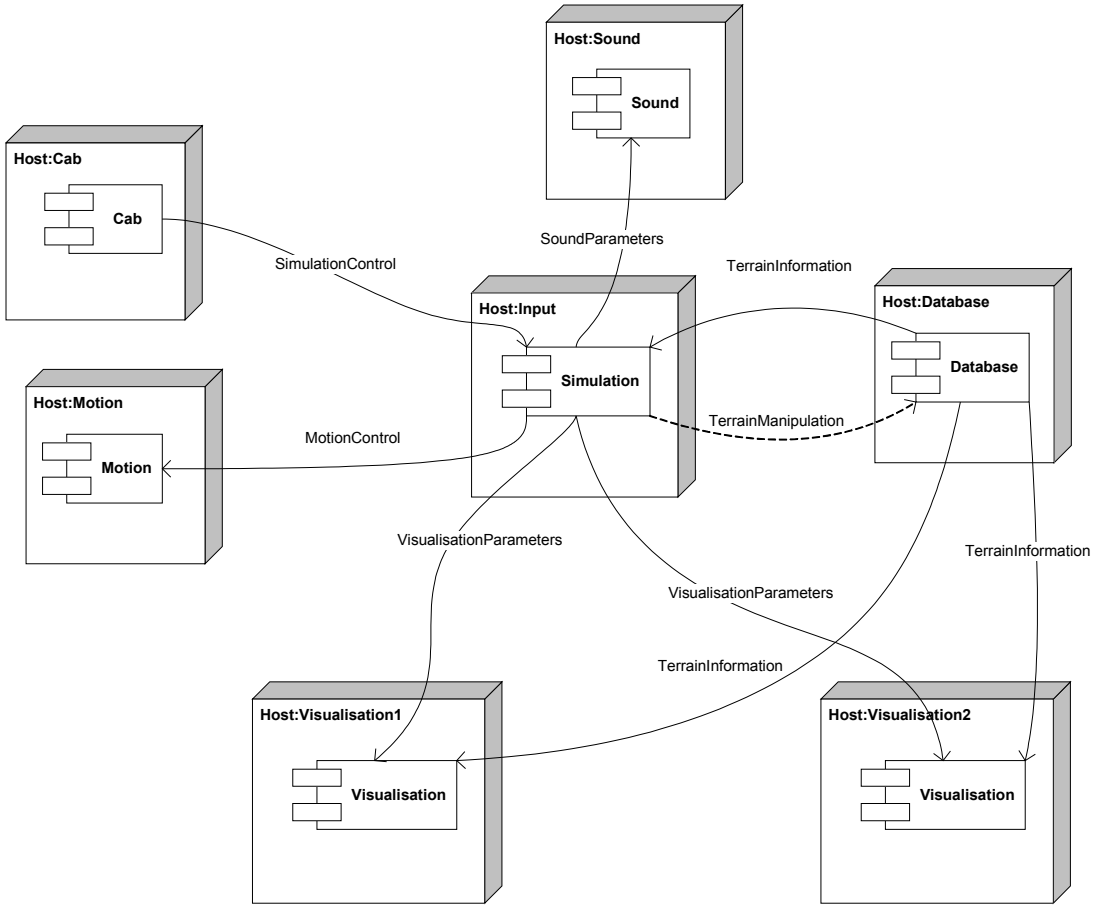


Abbildung 2.1: Vereinfachte Darstellung der aktuellen Konfiguration am Baggersimulator

kann jedoch nur getestet werden, wenn diese auch auf derselben Maschine verfügbar sind. Um die Implementierung neuer Komponenten möglichst einfach zu gestalten, darf es aus deren Sicht also keinen Unterschied zwischen lokaler und verteilter Kommunikation geben.

Abbildung 2.2 zeigt u.a. den DirectInputMapper, eine in C# implementierte Eingabekomponente von Robert Bürger, die mit Hilfe von DirectX verschiedenste Standardeingabegeräte wie Tastaturen, Mäuse, oder Joysticks in das Simulations- und VR-System integriert. Um diese für eine neue Anwendung nutzen zu können, muss der Entwickler lediglich die benötigten Eingabekanäle definieren indem er einer Taste, einem Knopf oder einer Bewegungsachse eine Kanalnummer zuordnet und festlegt, auf welchen Wertebereich die von der Hardware signalisierten Eingabeereignisse abgebildet werden sollen.

Mit Hilfe des DirectInputMappers, der Visualisierung von Joscha Metze und einer generischen Simulationskomponente die im Rahmen dieser Arbeit entstanden ist, wurde eine Vielzahl neuer Anwendungsfelder erschlossen. In Kapitel 6 wird darauf näher eingegangen.

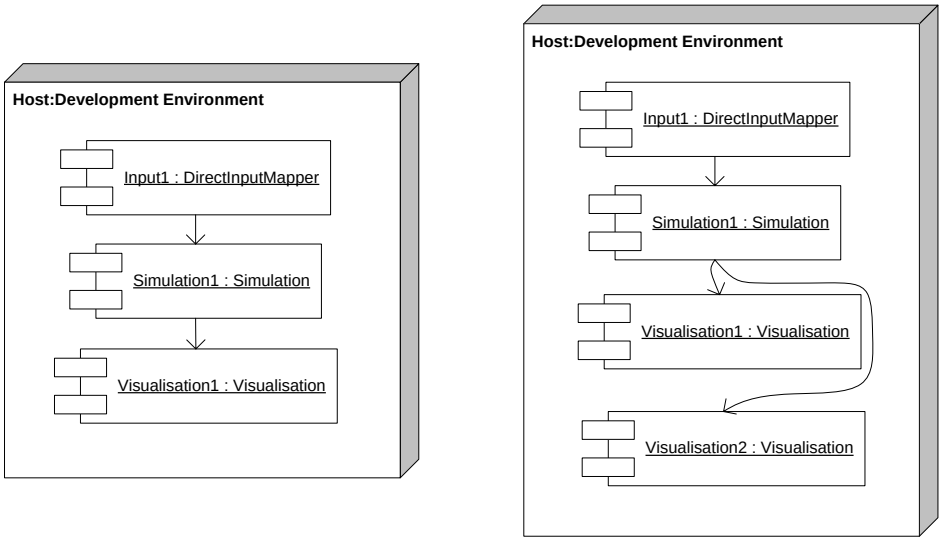


Abbildung 2.2: Entwicklung neuer Komponenten

2.2.3 Kontrollzentrum

Momentan gibt es noch keine Komponente, die die bequeme Verwaltung des verteilten Systems gestattet. Für diese Aufgabe wurde in [NMW05] ein spezielles Modul, das Kontrollzentrum (ControlCentre), konzipiert. Bei der prototypischen Implementierung durch Benny Wegener wurde aber festgestellt, dass die von der Netzwerkschicht angebotene Schnittstelle teilweise erweitert werden muss. Im folgenden werden deshalb der Entwurf dieser Anwendung und die daraus entstehenden Anforderungen an die Netzwerkschicht erläutert.

Um die Zusammenarbeit der einzelnen Komponenten zentral koordinieren zu können wurde ein Zustandsautomat definiert, den jedes Modul implementieren muss (Abbildung 2.3).

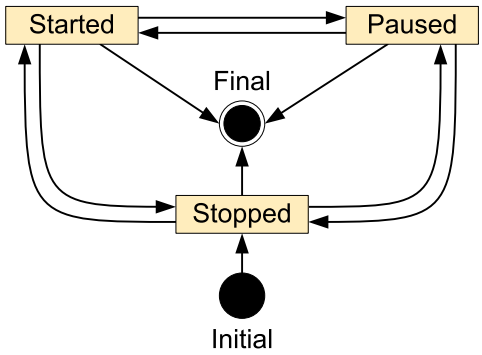


Abbildung 2.3: Zustandsautomat einer Komponente

Initial befinden sich alle Komponenten im Zustand „Stopped“, in dem sie ausschließlich auf weitere Anweisungen warten. Im Zustand „Started“ verrichtet die Komponente ihre Aufgabe. Der Zustand „Paused“

ist abhängig von der Art der Komponente: ein Eingabemodul sollte Eingaben ignorieren, eine Simulation sollte nach dem aktuellen Simulationsschritt anhalten, eine Visualisierung könnte weiterlaufen oder einen Bildschirmschoner anzeigen.

Damit die Zustandsübergänge über das ControlCentre ausgelöst werden können, ist zunächst eine globale Sicht auf das Gesamtsystem nötig, d.h. die physische Verteilung aller beteiligten Module muss erfasst werden können und jede Instanz muss einzeln adressierbar sein. In einer statischen Konfiguration, bei der jede Instanz immer auf dem selben Rechner ausgeführt wird, ist das nicht problematisch. Mechanismen, die die physische Verteilung in einer dynamischen Umgebung zur Laufzeit bestimmen können, fehlen jedoch in der bisherigen Implementierung der Netzwerkkommunikationsschicht.

Neben der globalen Sicht auf die eben definierten Ausführungszustände sollte das ControlCentre auch eine globale Sicht auf Ausführungsfehler ermöglichen. Ohne eine zentrale Auswertung von Fehlern könnte es aufwändig sein, am Gesamtverhalten des Systems festzustellen, welche Komponente einen Fehler verursacht hat. Um auch noch das Auffinden der Fehlerursachen zu erleichtern, wurden die Fehlerstufen „debug“, „info“, „warning“, „error“ und „critical“ festgelegt, und somit die Basis für die zentrale Protokollierung der Aktivitäten aller Instanzen geschaffen.

Weiterhin soll das ControlCentre eine spezielle Scripting-Schnittstelle anbieten, mit der Scripts zur Laufzeit auf den Modulen abgesetzt und zur Ausführung gebracht werden können. Im Wesentlichen sollen damit Konfigurationsparameter wie z.B. die Auflösung einer Visualisierungsinstanz manipulierbar gemacht werden.

Um die Administration einzelner Module, aber auch ganzer verteilter Anwendungen zu vereinfachen, wurde ein Konfigurationsmanagmentkonzept entworfen, bei dem die Ausgangsparameter aller verwendeten Module in so genannten „Workspaces“ organisiert sind. Dazu wurde ein generisches Beschreibungsformat definiert, mit dem sowohl die Verteilung (siehe auch 3.1.3), als auch die komponentenspezifischen Konfigurationsparameter festgelegt werden können. Da das ControlCentre das einzige Modul ist, welches eine globale Sicht auf das gesamte System hat, muss es zudem das Workspacemanagement über die Scripting-Schnittstelle übernehmen.

Abbildung 2.4 soll die Aufgaben des ControlCentres noch einmal zusammenfassen.

2.2.4 Verteilte Simulation

Grundsätzlich soll die beliebige Verteilung der Instanzen einer Simulations- und VR-Anwendung auf eine variable Anzahl von Rechnern möglich sein. In der Praxis wurden bisher mehrere Visualisierungen auf mehrere Standard-PCs verteilt, um so die höchstmögliche visuelle Qualität zu erreichen. Theore-

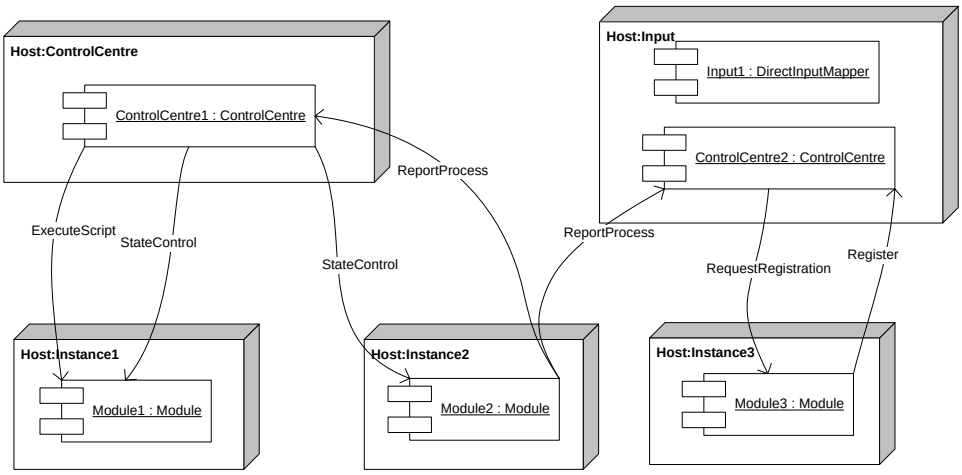


Abbildung 2.4: ControlCentre

tisch wären jedoch auch andere Konfigurationen denkbar, z.B. eine ressourcenhungrige Simulation, die parallelisiert werden muss. Parallele, verteilte Simulationen sind seit Jahren Forschungsgegenstand. Am erfolgreichsten scheint der vor militärischem Hintergrund entstandene IEEE-Standard für „Distributed Interactive Simulation“ (IEEE Std 1278.1a-1998, siehe [iee98]) zu sein. Da dieses Problem momentan im Umfeld des Maschinensimulators keine Rolle spielt, soll bei beim Entwurf der Netzwerkkommunikationsschicht auch nicht näher darauf eingegangen werden. Sollte dennoch irgendwann ein solches System entwickelt werden, wäre zunächst zu prüfen, ob die angebotenen Funktionen der Netzwerkschicht eine Synchronisation auf Anwendungsebene erlauben.

2.2.5 Multiplayer-Spiel

Wie beim Maschinensimulator können auch bei einem Spiel einzelne Komponenten auf verschiedene eigenständige Rechner verteilt werden. Die damit einhergehende Leistungssteigerung ermöglicht es, physikalisches Verhalten in Echtzeit zu berechnen und somit neuartige Spielkonzepte umzusetzen (Abbildung 2.5). Ein Beispiel wird in Kapitel 7 gezeigt.

2.3 Abgeleitete Anforderungen an die Kommunikationsschicht

Aus der Szenarioanalyse lassen sich mehr oder weniger offensichtliche Anforderungen an die Nachrichtenübermittlung ableiten[1]:

- Performanz: Um zu klären, welche Aspekte der Leistungsfähigkeit der Netzwerkschicht für das

[1]Die Begriffsdefinitionen wurden größtenteils aus [CDK01] entnommen und übersetzt. Andere Autoren verwenden z.T. abweichende Definitionen.

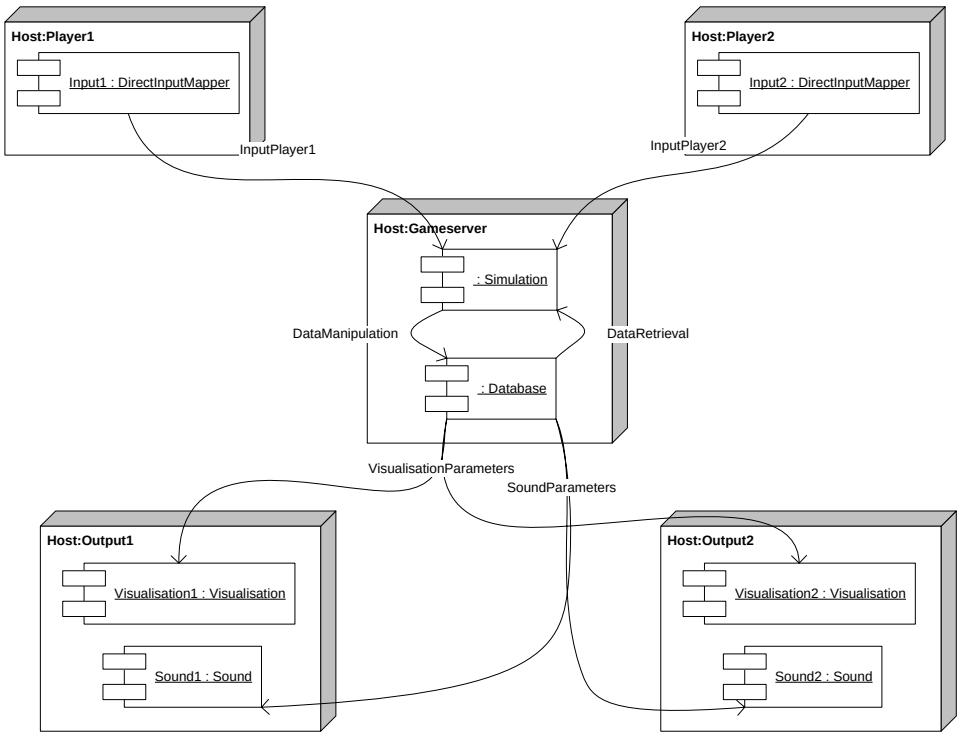


Abbildung 2.5: Multiplayer-Spiel mit Echtzeit-Physiksimulation

Simulations- und VR-System von besonderer Bedeutung sind, sollen zunächst die relevanten Begriffe und Zusammenhänge geklärt werden.

- Latenz: Die Latenz beschreibt die Zeit die vergeht, bis nach dem Absenden der Nachricht erstmals Daten auf der Empfängerseite verfügbar werden. Diese Verzögerung kommt durch den Netzwerkstack und die Laufzeit im Übertragungsmedium zustande.
- Datenübertragungsrate: Die Datentransferrate ist die Geschwindigkeit, mit der Daten zwischen zwei Computern ausgetauscht werden können, nachdem die Übertragung begonnen hat.
- Nachrichtenlaufzeit: Die Zeit, die benötigt wird um eine Nachricht zu übertragen, berechnet sich wie folgt:

$$Nachrichtenlaufzeit = Latenz + \frac{Nachrichtenlänge}{Datenübertragungsrate}$$

- Round Trip Time: Die Round-Trip-Zeit ist die Summe aus der Laufzeit einer Nachricht von der Quelle zum Ziel und der Laufzeit der Antwort vom Ziel zur Quelle.
- Antwortzeit[2]: Die Antwortzeit, auch als Gesamtverzögerung bezeichnet, setzt sich aus der Round Trip Time und der Verarbeitungszeit auf dem Zielrechner (Server Delay) zusammen.

[2] nach [Pac02]

Von diesen Performanzkriterien stellt sich die Latenz als am wichtigsten heraus, weil, mit Ausnahme der Datenübertragungsrate, alle anderen Größen von ihr abhängen. Damit ist sie sowohl für Nachrichten ohne Rückgabewert, als auch für Nachrichten zu denen eine Antwort erwartet wird von höchster Relevanz, denn hohe Verzögerungen verursachen nicht tolerierbare zeitliche Diskontinuitäten. Die Datenübertragungsraten hingegen haben in lokalen Netzen mittlerweile ein sehr hohes Niveau erreicht, so dass sich diese Größe erst bei sehr großen Nachrichten limitierend auf die Echtzeitfähigkeiten des Systems auswirkt.

- Zuverlässigkeit: Wie schon in [Met05] dargestellt, wird die Zuverlässigkeit der Übertragung durch Korrektheit, Reihenfolgetreue und Verlustrate bestimmt. Es hat sich gezeigt, dass für das Simulations- und VR-System sowohl zuverlässige, als auch unzuverlässige Übertragungskanäle wünschenswert sind. Die Nachrichten des ControlCentres z.B. müssen in jedem Fall zugestellt werden. Wie schnell die Übertragung stattfindet ist dabei zweitrangig. Wenn dagegen die Simulation die absoluten Positionen und Rotationen der animierten Objekte an die Visualisierung schickt, müssen enge Zeitgrenzen eingehalten werden. Der Verlust einzelner Nachrichten ist dabei tolerierbar, weil zeitnah neue Daten eintreffen werden.
- Multicasting: Neben der einfachen Nachrichtenübermittlung zwischen zwei Modulen muss es außerdem möglich sein, Nachrichten auch an mehrere Module gleichzeitig zu schicken. Generell könnte man zwar dieselbe Nachricht nacheinander an alle Zielmodule senden (serielles Unicasting), aber mit steigender Nachrichtenlaufzeit und Empfängerzahl driften die Empfangszeiten immer weiter auseinander. Wenn echte Gleichzeitigkeit gefordert ist, also z.B. bei der Projektion einer bewegten Szene in eine fünfwandige CAVE, für die zehn Visualisierungsinstanzen benötigt werden, ist dieser Ansatz aber unzureichend. Bessere Lösungen versprechen Multicasting (1:n) oder Broadcasting (1:all). Beide Technologien erlauben das gleichzeitige Versenden von Nachrichten an mehrere Empfänger. Multicasting wäre wünschenswert, weil Nachrichten damit exklusiv an eine festgelegte Gruppe von Empfängern gesendet werden können. Eine Broadcast-Nachricht hingegen muss zunächst von allen Knoten eines Subnetzes akzeptiert werden, damit diese entscheiden können, ob sie die Nachricht verarbeiten sollen oder nicht. Der effizienteren Ausnutzung der Bandbreite beim Multicasting steht allerdings die geringere Verfügbarkeit der benötigten Netzwerkhardware gegenüber.
- Gleichlaufschwankungen (Jitter): Bezogen auf Netzwerke ist mit Jitter die Varianz der Latenz oder der Datenübertragungsrate gemeint. Da der reibungslose Betrieb eines VR-Systems harte Echtzeitanforderungen stellt, machen sich Gleichlaufschwankungen leicht bemerkbar. Dieses Problem kommt am deutlichsten bei der Verarbeitung von Eingaben zu tragen. Das liegt zum einen daran, dass Eingabenachrichten unter Umständen nicht verworfen werden können, diskrete Ereignisse

wie z.B. Mouseclicks könnten sonst unregistriert bleiben, zum anderen wirken variable Verzögerungen bei der Steuerung besonders störend.

- Dienstgütekriterien: Dienstgütekriterien, besser bekannt als Quality of Service (QoS), beschreiben Kanaleigenschaften, die der Anwendung von der Transportschicht zugesichert werden. Neben oberen und unteren Grenzen für die verschiedenen Performanzkriterien sollen auch minimal und maximal zulässige Schwankungen vereinbart und garantiert werden können. Da verteilte Echtzeitanwendungen wenig tolerant gegenüber Schwankungen der Kanalqualität sind, wären zugesicherte Übertragungseigenschaften eine echte Bereicherung. In der Praxis sind QoS-fähige Netzwerke allerdings noch selten. Die beste Lösung bieten ATM-Netzwerke, wobei die Hardware auf Grund mangelnder Verbreitung sehr teuer ist. Einen Kompromiss könnten die Ansätze Integrated Services (IntServ; [RFCc]) und Differentiated Services (DiffServ; [RFCa]), die beide auf IP aufsetzen, darstellen (siehe auch [Kap03]). Zwar werden auch hier spezielle Router benötigt, diese könnten jedoch auf Grund der hohen Marktdurchdringung von Ethernet in Zukunft günstiger werden.

2.4 Programmiermodell und Aufrufsemantik

Zu den Aufgaben einer Middleware gehört es, die Heterogenität der darunterliegenden Programmiersprachen, Betriebssysteme, Hardwarekomponenten und Netzwerke zu maskieren. Dazu ist es notwendig, ein einheitliches Programmiermodell zu definieren, dessen Abstraktionsniveau den Erwartungen der Anwendungsentwickler entspricht. Verschiedene Modelle haben in der Praxis weite Verbreitung erfahren und sollen deshalb kurz vorgestellt werden.

Der Remote Procedure Call (RPC) ist die einfachste und älteste Variante [rfcd] [rfce]. Ziel des RPC ist es, entfernte Prozeduraufrufe wie lokale Prozeduraufrufe (LPC) aussehen zu lassen. Dazu verpackt der Clientprozess die Aufrufparameter in Nachrichten, die vom entfernten Serverprozess entgegengenommen, interpretiert und auf Funktionsaufrufe abgebildet werden. Falls eine Antwort benötigt, wird sendet der Serverprozess auf die gleiche Weise eine Nachricht an den Client zurück. Die Umwandlung der Aufrufparameter in ein Nachrichtenformat wird als „Marshalling“ bezeichnet, die Interpretation der Nachrichten und das Abbilden auf Funktionsaufrufe heißt „Unmarshalling“. Für den Anwendungsentwickler sind beide Prozesse transparent.

Ursprünglich lief ein entfernter Aufruf immer synchron ab, d.h. der Client blockierte so lange, bis er eine Antwort vom Server erhielt. Diese Aufrufsemantik ist jedoch nicht immer angemessen, deshalb kamen später asynchrone Erweiterungen dazu, so dass der Client Aufrufe absetzen kann, ohne auf die Antworten warten zu müssen. Statt dessen werden Callback-Funktionen definiert, die vom RPC-System

aufgerufen werden sobald die Antworten eingetroffen sind. Die wichtigsten Implementierungen des RPC sind ONC-RPC (Open Network Computing RPC), auch SunRPC genannt, und DCE-RPC (Distributed Computing Environment). ISO-RPC (ISO/IEC CD 11578 N6561) kann als gescheitert betrachtet werden. Da C# jedoch nur von der kommerziellen Implementierung der Distinct Corporation unterstützt wird, proprietäre Software aber weitgehend vermieden werden soll, ist diese Lösung nicht optimal.

Bei den weitergehenden, objektorientierten Ansätzen kommunizieren nicht mehr Clientprozesse mit Serverprozessen, sondern Objekte im Adressraum eines Clients mit Objekten im Adressraum eines Servers. Der einfachste Vertreter dieser Kategorie ist Java RMI (Remote Method Invocation [rmi]), universeller ist der von der OMG vorgestellte Standard CORBA (Common Object Request Broker [cor]). Die damit einhergehende Vereinfachung des Programmiermodells wird allerdings mit einem hohem Performanzverlust erkauft. Diese Lösungen sind schon alleine deshalb für die verteilte Echtzeitsimulation unbrauchbar.

Daneben gibt es noch Message-Queue-Systeme, manchmal auch Message-Passing-Systeme genannt. Hier werden Nachrichten von einer Warteschlange auf Clientseite zu einer Warteschlange auf Serverseite übertragen. Neben vielen kommerziellen Implementierungen gibt es eine Vielzahl an freien Alternativen. Diese Systeme wurden für die asynchrone Datenübertragung in Umgebungen mit eingeschränkter Konnektivität entwickelt, weshalb synchrone Aufrufmechanismen fehlen.

Zusammenfassend lässt sich feststellen, dass ein RPC-System die Anforderungen an die Netzwerkkommunikationsschicht am besten erfüllen kann. Neben blockierenden Funktionsaufrufen sollten aber grundsätzlich auch asynchrone Aufrufe möglich sein. Weiterhin soll der Anwendungsentwickler festlegen können, welchen Wert eine Funktion zurückgeben soll, wenn ein Timeout aufgetreten ist. Die Netzwerkschicht kennt die Bedeutung der Rückgabewerte nicht, deshalb kann dort auch kein Fehlerwert festgelegt werden.

Zusätzliches Optimierungspotential könnte durch die Unterstützung von Multithreading erschlossen werden. Der Anwendungsentwickler könnte dann synchrone Funktionsaufrufe mit Rückgabewert durch die Verwendung von mehreren Threads parallelisieren. Dadurch ließen sich die Antwortzeiten verkürzen, weil neue Anfragen bereits gestellt werden könnten, während andere Threads noch auf ihre Antworten warten. n Aufrufe können so maximal um den Faktor $\min(1 + \frac{Nachrichtenlaufzeit}{Ausführungszeit}; n)$ beschleunigt werden[3]. Aus der Formel ist ersichtlich, dass wesentliche Verbesserungen nur möglich sind, wenn die Laufzeit im Verhältnis zur Ausführungszeit (Server Delay) groß ist. Typischerweise sind in einem LAN die Nachrichtenlaufzeiten aber relativ gering, so dass in diesem Fall nur dann eine Verbesserung zu erwarten ist, wenn die Abarbeitung der Anfragen auf Serverseite sehr schnell geschieht und die Nachrichten relativ groß sind.

[3]Das Minimum wird gebildet, weil pro Aufruf maximal die Laufzeit eingespart werden kann.

2.5 Evaluation der bestehenden Netzwerkkommunikationsschicht

Die eingehende Betrachtung der bestehenden Netzwerkbibliothek ergab, dass die wichtigsten Anforderungen zum größten Teil erfüllt werden. Vor allem bei der Administrierbarkeit und Erweiterbarkeit gibt es aber noch Defizite. Im folgenden sollen diese Probleme, aber auch kleinere Schwächen bei Entwurfs- und Implementierungsdetails, stichpunktartig aufgeführt werden:

- Verwaltung, Konfiguration und globale Zustandskontrolle schwierig oder nicht möglich:
  - keine Mechanismen um zur Laufzeit die Verteilung der Komponenten zu ermitteln (2.2.3)
  - globale Zustandskontrolle noch gar nicht berücksichtigt (2.2.3)
  - statische Konfiguration relativ unflexibel; Anwendung kann sie nur lesen, nicht aber verändern und speichern (3.1.3)
- Erweiterbarkeit problematisch:
  - keine festgelegte Schnittstelle zur Transportschicht, statt dessen nur eine Klasse für alle Transporttypen (→ `if(tcp)... else if(udp)... else if(broadcast)...`) (3.2)
  - kaum Dokumentation, teilweise missverständliche Namensgebung
- Aufrufsemantik und Programmiermodell könnten flexibler gestaltet sein:
  - kein funktionspezifischer Fehlerrückgabewert bei automatisch generierten Schnittstellen definierbar (3.1.1)
  - default timeout (1000 ms) für alle Funktionsaufrufe hardcoded (3.1.1 und 3.1.3)
  - bei Broadcasting keine selektiven Antworten möglich, sondern Anwort auch immer Broadcast (3.2.1)
- Schwächen in der Implementierung:
  - Sicherheitsaspekte nicht berücksichtigt → eher für LAN als für WAN geeignet (7)
  - keine Socketpuffergrößen einstellbar, mit angepassten Puffern wäre Leistungssteigerung möglich (3.2.1)
  - unnötige Konvertierung von Host Byte Order in Network Byte Order und zurück (3.2.2)
  - keine Auswahl unter mehreren Netzwerkkarten möglich → kein spezifisches Bind (4.2)
  - UDP-Implementierung nur für kleine Nachrichten geeignet; zudem unvorhersagbares Verhalten bei Pufferüberlauf (4.3); weiterhin wird nach jeder gesendeten Nachricht auf eine Empfangsbestätigung gewartet und die Nachricht bei Ausbleiben derselben wiederholt → hohe Latenz, obwohl gerade die UDP-Kommunikation kurze Verzögerungen bieten soll!

- Teile der Implementierung haben sich als überflüssig herausgestellt:
  - je ein Socket zum Senden- bzw. Empfangen
  - CRC für alle Nachrichten überflüssig, da bereits in TCP und UDP realisiert
  - ursprünglich mehrere verschiedene Dienste pro Peer angedacht, aber unvollständige Umsetzung → Feature nicht nutzbar, aber Codekomplexität erhöht

3 Entwurf

Die Anforderungsanalyse, die Evaluation der bestehenden Netzwerkbibliothek und die Erfahrungen in der Praxis haben gezeigt, dass beim ursprünglichen Entwurf wichtige Aspekte nicht beachtet wurden und die Implementierung an vielen Stellen verbessert werden könnte. Deshalb wurde ein neuer Entwurf erarbeitet, der sich in den Grundzügen zwar noch an die alte Netzwerkkommunikationsschicht anlehnt, die Benutzung aber vereinfacht und die Transportschicht austauschbar macht.

3.1 Sicht auf die Kommunikationsschicht

Wie in 2.4 festgestellt, soll das Programmiermodell der Netzwerkommunikationsschicht an dem des Remote Procedure Call angelehnt sein. Um dieses Abstraktionsniveau zu erreichen, muss es möglich sein, die Schittstelle zur Kommunikationsschicht um anwendungsspezifische Funktionen zu erweitern. Dazu sind Mechanismen erforderlich, die aus der Beschreibung einer Funktion entsprechende Bibliotheken für das Marshalling und Unmarshalling erzeugen können (3.1.1).

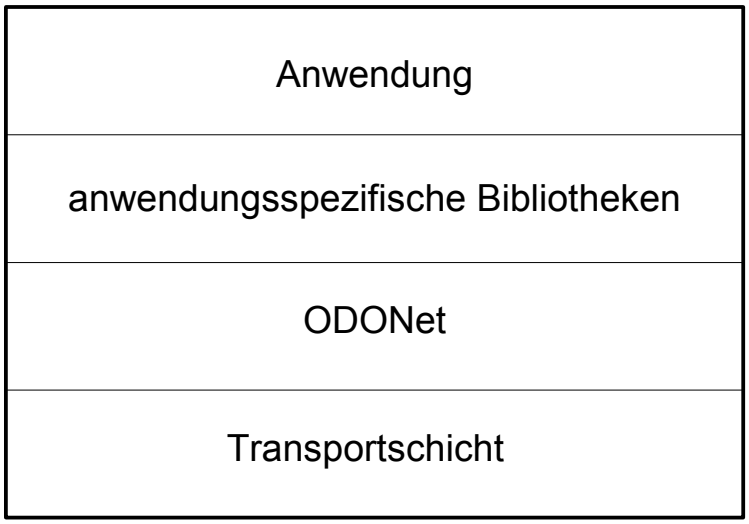


Abbildung 3.1: Schichtenarchitektur

Neben den exportierten Funktionen ist es für den Anwendungsentwickler vor allem relevant, wie er die Kommunikationsendpunkte konfigurieren und verwenden kann. Bisher waren die Konfigurations-

möglichkeiten stark eingeschränkt, daher musste ein flexibleres Konzept erarbeitet werden. Die einfache Benutzung der alten Netzwerkbibliothek sollte durch eine Peer-to-Peer-Architektur ermöglicht werden, in der Praxis konnte sich dieser Ansatz aber nicht durchsetzen, weswegen jetzt wieder ein einfaches Client-Server-System realisiert werden soll (3.1.2).

3.1.1 Unterstützte Funktionen und Datentypen

Wie beim RPC soll die Netzwerkkommunikationsschicht das Marshalling und Unmarshalling transparent machen, so dass der Anwendungsentwickler Funktionen aufrufen kann, ohne sich um das Zusammenstellen, Versenden und Interpretieren von Nachrichten kümmern zu müssen. Dazu gibt es wie bisher die Möglichkeit, die Schnittstelle zur Netzwerkschicht durch die automatische Generierung spezialisierter Kommunikationsendpunkte zu erweitern. Um diese schnell und einfach zur Verfügung stellen zu können, muss zunächst eine Schnittstellenbeschreibungssprache (Interface Definiton Language) festgelegt werden. Der Programmierer spezifiziert damit die von ihm benötigten Funktionen, welche dann automatisch in C++- oder C#-Quellcode übersetzt und anschließend kompiliert werden (siehe 4.6).

Die bisher verwendete Interface Definition Language war nicht optimal, deshalb wurde sie durch ein neues Schema ersetzt. Durch die Verwendung von XML-Attributen an Stelle von XML-Elementen wird die Beschreibung einer Schnittstelle kompakter und lässt sich zudem einfacher parsen und übersetzen. Ein kleines Beispiel soll die neue Schnittstellenbeschreibungssprache illustrieren (Listing 3.1).

Der Entwickler beschreibt also die Signatur der benötigten Funktionen und optional einen Timeout-Wert der angibt, wie viele Millisekunden ein Aufruf dauern darf. Liegt nach Ablauf dieser Zeit kein Ergebnis vor, wird der Fehlerwert zurückgegeben. Dieser muss für alle Funktionen, die einen Rückgabewert erwarten, angegeben werden. Ein Default-Fehlerwert kann nicht festgelegt werden, weil die Kommunikationsschicht die Bedeutung der Rückgabewerte nicht kennt. Deswegen muss der Programmierer selbst dafür sorgen einen Fehlerwert zu finden, der sich von regulären Rückgabewerten unterscheidet. Über das Attribut `"id"` wird ein anwendungsspezifisches Protokoll definiert, das es ermöglicht, Funktionen und Nachrichten aufeinander abzubilden. Diese Identifikationsnummer muss innerhalb einer Schnittstellendefinition eindeutig sein.

Für größtmögliche Flexibilität sollten möglichst viele Datentypen bei der Definition der Funktionen erlaubt sein. Die Evaluation der bestehenden Netzwerkkommunikationsschicht ergab jedoch, dass die Verwendung der primitiven Datentypen `short` und `long` zu Problemen führen wird, sobald Prozessorarchitekturen unterschiedlicher Wortbreite eingesetzt würden. Da an diesen Datentypen kein dringender Bedarf besteht, werden sie nicht weiter unterstützt. Es bleiben also `byte`, `int`, `float`, `double`,

Listing 3.1: Einfache Schnittstellenbeschreibung

```
<?xml version="1.0" encoding="utf-8" ?>
<messagetable name="example_interface">
  <!-- a value for attribute "error" is mandatory if the returntype is different from void -->
  <message name="SetInt" id="1" error="ERROR">
    <returntype type="string" />
    <parameter type="int" name="valueID" />
    <parameter type="int" name="value" />
  </message>
  <!-- for this method, Int32.MinValue is chosen to indicate an error -->
  <message name="GetInt" id="2" error="0x80000000">
    <returntype type="int" />
    <parameter type="int" name="valueID" />
  </message>
  <message name="SetIntArray" id="3">
    <returntype type="void" />
    <parameter type="int" name="valueID" />
    <parameter type="int[]" name="value" />
  </message>
  <!-- the timeout in ms is optional; default is according to stub configuration -->
  <!-- "0|0|0" describes a three-dimensional array filled with zeros -->
  <!-- in this case, it is an integer array as specified by the element "returntype" -->
  <message name="GetIntArray" id="4" timeout="1000" error="0|0|0">
    <returntype type="int[]" />
    <parameter type="int" name="valueID" />
  </message>
</messagetable>
```

`(std::)string`[1]. Die zugehörigen Arraydatentypen sind unter C++ jetzt immer als getypte Vektoren (`std::vector`) realisiert. Neu ist die Möglichkeit String-Arrays zu übertragen und dass jeder Funktion ein individueller Timeout- und Fehlerrückgabewert zugewiesen werden kann. Alle unterstützten Datentypen sind in Tabelle 3.1 zusammengefasst.

An dieser Stelle sei außerdem noch auf die `XData`-Klasse der Lore-Bibliothek hingewiesen, die neben impliziten Typkonvertierungen zwischen vielen primitiven Datentypen auch die Serialisierung strukturierter Daten in Strings der Form "`value1|value2|...|valueN`" beherrscht. Bei der Deserialisierung werden diese pipe-separierten Strings dann wieder auf den passendsten Array-Datentyp abgebildet. Am Lehrstuhl für Computergrafik wird diese Funktionalität häufig dafür verwendet, um über XML-Dateien programmiersprachenunabhängig Daten zwischen verschiedenen Anwendungen auszutauschen. Ein Beispiel soll verdeutlichen wie diese Funktionalität genutzt werden kann:

[1] Strings werden ASCII-codiert. Werden Zeichen aus einer 8-bit Erweiterung verwendet, also z.B. deutsche Umlaute, kann es abhängig von der Plattform und den Lokalisierungseinstellungen zu Problemen kommen.

primitive Datentypen		Array-Datentypen	
C#	C++	C#	C++
byte	typedef byte unsigned char*	byte[]	std::vector<byte>
int	int	int[]	std::vector<int>
float	float	float[]	std::vector<float>
double	double	double[]	std::vector<double>
string	std::string	string[]	std::vector<std::string>

Tabelle 3.1: Unterstützte Datentypen

Der Anwendungsentwickler möchte einen dreidimensionalen Vektor doppelter Genauigkeit verschicken. Dazu definiert er folgendene Funktion (siehe auch 4.6):

```
//C#
void SendVector3D(double[] vector);
//C++
void SendVector3D(std::vector<double>);
```

Der Aufruf der Funktion könnte dann wie folgt aussehen:

```
//C#
clientStub.SendVector3D((Lore.Datatypes.XData)"0|.23|3.14");
//C++
clientStub->SendVector3D(lore::XData("0|.23|3.14"));
```

3.1.2 Client-Server-Architektur

Die im usprünglichen Entwurf geplante Peer-to-Peer-Architektur wurde nur unvollständig umgesetzt: Die automatische Generierung spezialisierter Bibliotheken unterstützte nur das Client-Server-Prinzip, so dass gleichberechtigte Kommunikationspartner (Peers) von Hand programmiert werden mussten. Auch in den Konfigurationsdateien und im Quellcode wurden die Kommunikationsendpunkte nicht konsequent als Peers, sondern als Client oder Server bezeichnet. Teilweise herrschte dabei unter den Entwicklern Uneinigkeit, welcher Kommunikationsendpunkt als Server und welcher als Client zu bezeichnen sei.

Um die Rollenverteilung unmissverständlicher zu machen, wurde ein Proxy-Konzept umgesetzt. Wie beim RPC werden die Proxies (Stellvertreterobjekte) als Stubs bezeichnet. Der ServerStub ist dabei ein Proxyobjekt, über das ein Client Serverfunktionen aufrufen kann. Auf Serverseite werden über den ClientStub Antwortnachrichten verschickt. Dieser Ansatz erscheint sinnvoll, weil die Verteilung der Kom-

ponenten ohnehin nicht ganz transparent ist, der Anwendungsentwickler also weiß, ob er eine entfernte oder eine lokale Funktion aufruft. Abbildung 3.1.2 verdeutlicht die gewählte Architektur.

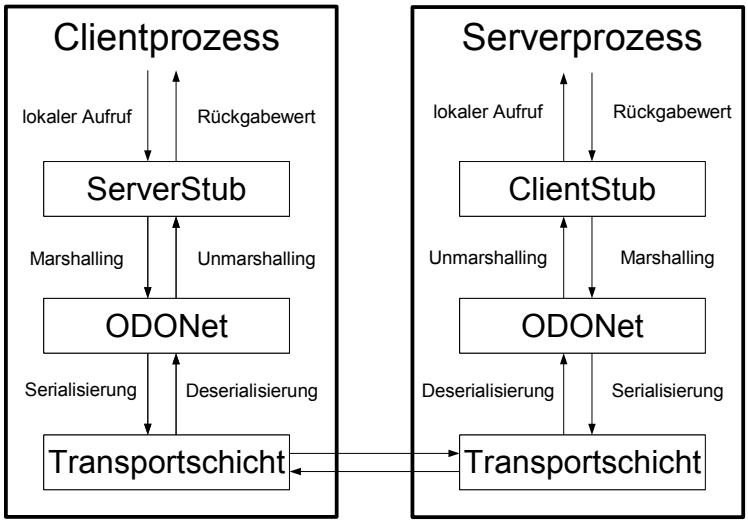


Abbildung 3.2: Client-Server-Architektur

Um zukünftigen Missverständnissen vorzubeugen, können nur ServerStubs, also Objekte auf Clientseite, einen Kommunikationsvorgang initiieren, wohingegen ClientStubs ausschließlich Antwortnachrichten verschicken können.

Außerdem kann ein Server nur noch einen Dienst anbieten. Soll eine Anwendung mehrere Dienste bereitstellen, muss pro Dienst ein ClientStub instanziiert werden. Dadurch steigt zwar der Speicherbedarf ein wenig, die Komplexität der Netzwerkbibliothek wird allerdings reduziert, so dass eine Leistungssteigerung möglich ist.

3.1.3 Konfiguration

Um die einzelnen Module möglichst einfach zu neuen verteilten Anwendungen verknüpfen zu können, müssen sie flexibel konfigurierbar sein. Neben der physikalischen Verteilung und den Verbindungseigenschaften gibt es anwendungsabhängig weitere Parameter, die eingestellt werden können sollen.

Die Konfiguration der Kommunikationsendpunkte geschieht, anders als beim RPC, über XML-Dateien. Beim RPC kommen ein Verzeichnisdienst (Directory Service) und pro Host ein RPC-Dämon zum Einsatz [rfcb]. Dadurch wird die Verteilung der Komponenten zwar völlig transparent, auf der anderen Seite wird jede Anwendung dadurch von der Existenz eines Verzeichnisdienstes abhängig. Auf die Verteilungstransparenz kann aber zugunsten der Einfachheit verzichtet werden, da das Simulations- und VR-System normalerweise ein geschlossenes System unter der Kontrolle weniger Administratoren ist, und es nicht primär dazu gedacht ist, dezentral zu arbeiten.

Die Verwendung von XML bietet sich hier besonders an, weil sich die Konfigurationdateien damit sowohl von Hand, als auch aus einem Programm heraus, erstellen oder bearbeiten lassen. XML wird zwar schon länger zur Konfiguration bestimmter Komponenten des Baggersimulators verwendet, das Schema wurde jedoch komplett überarbeitet. An Stelle des Formats von Joscha Metze, mit dem ausschließlich Netzwerkverbindungen und Visualisierungsparameter festgelegt werden konnten, tritt eine generische Beschreibung mit der alle Module konfiguriert werden können.

Eine Konfigurationsdatei kann nach dem neuen Schema mehrere Sektionen beinhalten, in denen Objekte mit ID und Typinformation enthalten sind. Die ID ist als eindeutiger Name des Objekts zu vergeben, Typinformationen sind je nach Anwendung vom Entwickler frei wählbar und erlauben eine flexible Kategorisierung der Objekte. Die eigentlichen Parameter werden den Objekten als Elemente mit Namen und Wert zugewiesen. Sektionen können verwendet werden, um z.B. die Netzwerkkonfiguration von der Konfiguration der Kameras einer Visualisierung zu trennen.

Den Zugriff auf die Konfigurationsdateien ermöglicht die Klasse Configuration im Namensraum `Util` der Lore-Bibliothek. Nach der Instanziierung dieser Klasse mit dem Pfad zur Konfigurationsdatei als Konstruktorparameter können einzelne Konfigurationsobjekte (Klasse `ConfigObject`) anhand ihres Namens abgefragt werden. Alternativ können alle Konfigurationsobjekte, die bestimmten Bedingungen bezüglich Sektion, Typ oder Name genügen, ausgewählt werden. Danach müssen die Konfigurationsobjekte noch auf selbstdefinierte Klassen abgebildet werden, wobei für alle `attrib`-Elemente der Konfigurationsdatei eine öffentliche Variable mit dem selben Namen gesucht wird, die dann den Wert aus der Datei zugewiesen bekommt. Mit Hilfe der Klasse `Configuration` ist es nun auch möglich, neue Konfigurationsdateien anzulegen, Veränderungen an bestehenden Konfigurationen zu speichern oder verschiedene Konfigurationsdateien zusammenzuführen. Damit ist es nun u.a. möglich, die physikalische Verteilung der Module aus einer Anwendung heraus festzulegen.

Für die Konfiguration der Stubs wurde die Klasse `NetworkConfigEntry` entworfen, die Variablen für alle relevanten Netzwerkparameter enthält. Die Werte dieser Variablen werden aus dem entsprechenden Konfigurationsobjekt übernommen. Neben dem immer benötigten Verbindungstyp muss ein `NetworkConfigEntry`-Objekt meistens noch weitere Attribute wie Port oder Servername besitzen. Außerdem gibt es eine Vielzahl an optionalen Attributen, wie z.B. Default-Timeout oder Socketpuffergrößen. Um die Konfiguration der Netzwerkverbindungen von der verwendeten Netzwerktechnologie unabhängig zu machen, wurde das leere Interface `INetworkConfigEntry` definiert. So kann bei Bedarf eine neue Implementierung zur Bibliothek hinzugefügt werden, ohne dass Konstruktor- oder Aufrufparameter bei den Stubs oder den internen Klassen der Kommunikationsschicht geändert werden müssen.

An dieser Stelle soll nur eine sehr einfache Beispielkonfiguration (Listing 3.2) gezeigt werden, ein ausführlicheres Beispiel findet sich im Anhang (Listing A.1). Details sind der Anwenderdokumentation zu entnehmen.

Listing 3.2: Einfache Konfigurationsdatei

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration name="example_configuration">
  <section id="network">
    <!-- a TCP server that listens for connections on port 3000 -->
    <object id="ExampleClientStub" type="NetworkServer">
      <attr name="type">TcpServer</attr>
      <attr name="port">3000</attr>
    </object>
    <!-- a TCP client that connects to the server 141.76.71.139 -->
    <object id="ExampleServerStub" type="NetworkClient">
      <attr name="type">TcpClient</attr>
      <attr name="servernames">141.76.71.139</attr>
      <attr name="port">3000</attr>
    </object>
  </section>
</configuration>
```

3.2 Schnittstelle zur Transportschicht

Bisher gab es keine klar definierte Schnittstelle zur Transportschicht. Die Klasse `NetworkPeer` versendete die Nachrichten je nach aktueller Konfiguration über TCP, UDP oder Broadcast. Durch die dazu notwendigen Fallunterscheidungen wurden die Funktionen zum Senden und Empfangen sehr groß und damit schwer les- und erweiterbar. Auch der Code für die Initialisierung und die Zerstörung eines NetworkPeers wurde dadurch sehr unübersichtlich.

Um die Erweiterbarkeit hinsichtlich verschiedener Protokolle und Netzwerktechnologien zu verbessern, wurde eine Schnittstelle definiert, die eine geeignete Abstraktion zur darunterliegenden Transportschicht bieten soll (Abbildung 3.3). In der folgenden Beschreibung werden der Einfachheit halber nur die Bezeichnungen für C# verwendet, die äquivalenten Umsetzungen der Interfaces durch abstrakte Klassen in C++ heißen genauso, wobei an Stelle des `I` für `Interface` ein `Abstract` vorangestellt wird.

Als Basis für alle Klassen, die den Zugriff auf das Netzwerk ermöglichen sollen, wurde das Interface `INetworkPeer` definiert[2]. Ein NetworkPeer muss lediglich Ereignisse über empfangene Nachrichten

[2]Mit Peers sind in diesem Zusammenhang keine gleichberechtigten Kommunikationspartner gemeint. Der Begriff soll für beliebige Kommunikationsteilnehmer stehen, egal ob sie die Client- oder die Serverrollen einnehmen.

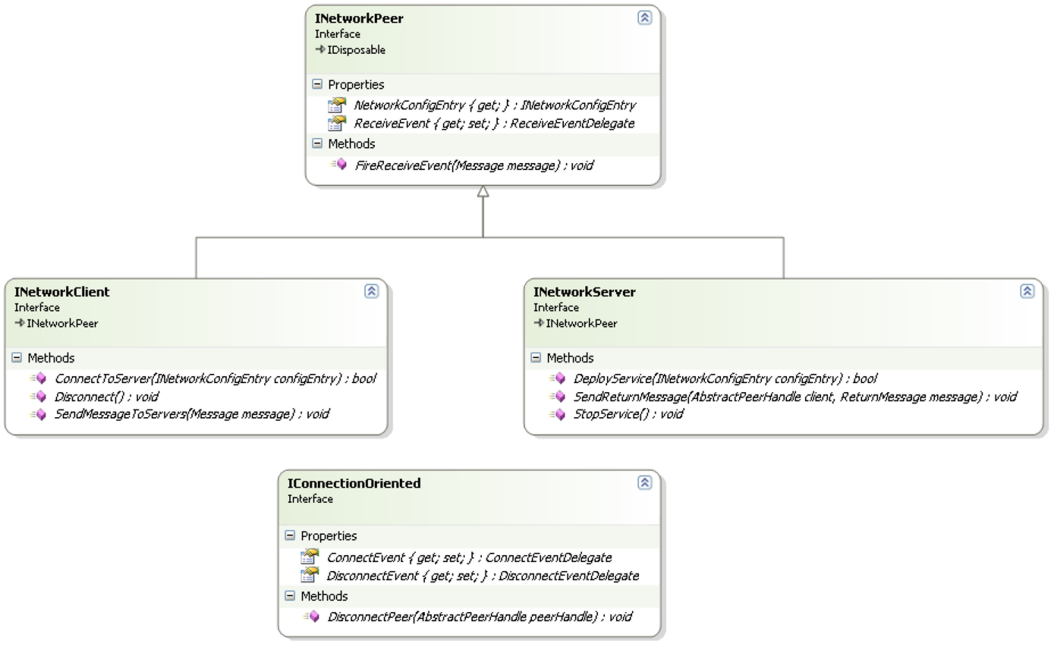


Abbildung 3.3: C#-Klassendiagramm der Schnittstelle zur Transportschicht

signalisieren und seine Konfiguration (`INetworkConfigEntry`) veröffentlichen.

Davon abgeleitet sind die Interfaces `INetworkServer` und `INetworkClient`. Ein NetworkClient muss Verbindungen zu Servern auf- und abbauen können und den Versand von Nachrichten ermöglichen. Ein NetworkServer muss Funktionen zum Bereitstellen und zur Beendigung eines Dienstes implementieren und Antwortnachrichten verschicken können. Die Beschränkung auf das Versenden von Antwortnachrichten dient der strengeren Durchsetzung einer klaren Trennung von Client- und Serverrolle. Initiator einer Kommunikation ist stets der Client. Ein Server kann nur auf Anfragen reagieren, niemals jedoch proaktiv Anfragen versenden.

Manche Anwendungen, insbesondere das Kontrollzentrum (siehe 2.2.3), müssen informiert werden, wenn eine neue Verbindung angenommen oder eine bestehende Verbindung beendet wurde. NetworkPeers die verbindungsorientiert arbeiten, müssen daher das Interface `IConnectionOriented` implementieren und Abonennten über Verbindungsereignisse informieren.

Die Parameter aller in diesen Interfaces definierten Funktionen und Ereignissen sind generisch. Es werden keine Annahmen über die Transportschicht oder die verwendete Netzwerktechnologie getroffen, so dass die Erweiterbarkeit auch jenseits von TCP/IP und UDP/IP sichergestellt ist.

3.2.1 Unterstützte Transporttypen

Wie in 2.3 dargestellt, soll die Netzwerkschicht sowohl zuverlässige, als auch unzuverlässige Kommunikation ermöglichen. Das Transmission Control Protocol (TCP) und das User Datagram Protocol (UDP) scheinen dafür geeignet. Zunächst soll verdeutlicht werden, was Zuverlässigkeit und Unzuverlässigkeit im Kontext dieser Protokolle bedeuten, um anschließend die Anforderungen an die Implementierung der verschiedenen NetworkPeers ableiten zu können.

TCP bietet als zuverlässiges Protokoll der Transportschicht die garantierte Zustellung und Reihenfolgetreue von Nachrichten. Erreicht wird dies durch die Retransmission von verloren gegangenen Segmenten und einer Flusssteuerung in Kombination mit einer Überlastkontrolle (congestion control).

Um die Zustellung sicherzustellen, muss der Empfänger dem Sender mitteilen, welche Segmente er empfangen hat. Bleibt eine solche Bestätigung für ein bestimmtes Paket für eine längere Zeit aus, wird es vom Sender neu übertragen. Das Timeout-Intervall wird dabei von TCP dynamisch an die Übertragungscharakteristik der Verbindung angepasst.

Die Flusssteuerung erfolgt mit dem Schiebefensterprotokoll (sliding window protocol) [Tan96]. Die Schiebefenstergröße bestimmt die Anzahl der Pakete, die der Sender versenden kann, ohne dass er dafür vom Empfänger eine Bestätigung erhalten hat. Zusätzlich zu einer Mitteilung über bereits empfangene Pakete, wird dem Sender vom Empfänger mitgeteilt, wie viele neue Daten dieser senden darf. Diese Flusskontrolle ist notwendig, weil ankommende Pakete verworfen werden müssten, falls die Empfangspuffer bereits voll sind. Die Wiederholung von Paketen würde in diesem Fall den Empfänger nur noch mehr überlasten.

Eng verbunden mit der Flusssteuerung ist die Überlastkontrolle [rfcf]. Aufgabe der Überlastkontrolle ist es, die Schiebefenstergröße[3] zu optimieren. Ist das Schiebefenster zu klein, wird die Bandbreite nicht voll ausgenutzt. Ist es zu groß, müssen sehr viele Segmente zwischengespeichert werden, um sie im Fehlerfall wiederholt übertragen zu können. Die Anpassung der Fenstergröße geschieht zunächst mit dem Slow-Start-Algorithmus: Initial darf nur ein Segment übertragen werden. Mit jeder Bestätigung über den erfolgreichen Empfang der zuletzt gesendeten Pakete, wird die Schiebefenstergröße bis zu einem bestimmten Maximalwert (meist 64kB) verdoppelt. Danach erhöht ein anderer Algorithmus, die TCP-Überlastvermeidung (congestion avoidance), die Größe des Fensters linear. Wird das Timeout-Intervall für eine Bestätigung überschritten, beginnt der Slow-Start-Algorithmus von neuem und drosselt dadurch die Senderate.

[3] Oft liest man im Zusammenhang mit der Überlastkontrolle von einem Überlastfenster (congestion window). Letzlich ist dies aber nur ein Wert, der die aktuell nutzbare Schiebefenstergröße limitiert. Deshalb soll „Schiebefenstergröße“ hier immer das Minimum von Schiebefenstergröße und Überlastfenstergröße bezeichnen.

Im Gegensatz zu TCP bietet UDP weder die garantierte Zustellung von Paketen, noch ihre Reihenfolgetreue. Auch Mechanismen zur Fluss- und Überlaststeuerung fehlen. Der Verwaltungsaufwand ist jedoch geringer, so dass gegenüber TCP kleinere Latenzen erreicht werden können. Beschränkt man die Kommunikation jedoch auf lokale, drahtgebundene Netze, ist der Begriff „unzuverlässige Kommunikation“ nicht mehr ganz zutreffend. Werden bestimmte Rahmenbedingungen eingehalten, kommen nahezu alle Pakete in der richtigen Reihenfolge beim Empfänger an.

Die maximal zulässige Bitfehlerrate beim Gigabit-Ethernet-Standard z.B. liegt bei 10^{-12} [Sub]. Statistisch kommt es also bei einer Paketgröße von 512 Byte erst nach 244 Millionen Paketen zu einem Fehler, der dann durch den Cyclic Redundancy Check innerhalb der Transportschicht erkannt wird und letztlich in einem verlorenen Paket resultiert. Üblicherweise gibt es in lokalen Netzen auch nur eine Route vom Sender zum Empfänger, so dass auch UDP-Pakete den Empfänger immer in der korrekten Reihenfolge erreichen.

Dramatischer kann sich das Fehlen der Fluss- und Überlaststeuerung auswirken. Kommen beim Empfänger mehr Pakete an als er verarbeiten kann, gehen sie unweigerlich verloren. Hiervon sind nicht nur die Kommunikationsendpunkte, sondern auch dazwischen liegende Switches oder Router betroffen. Solange eine Nachricht nicht auf mehrere Pakete verteilt werden muss, ist das Resultat eines Paketverlusts lediglich ein Timeout, auf den die Anwendung angemessen reagieren kann. Gehen jedoch Teile einer Nachricht verloren kann es passieren, dass nachfolgende Pakete falsch interpretiert werden, was im schlimmsten Fall zum Absturz der gesamten Anwendung führt.

Beide Protokolle sollten mit der Netzwerkbibliothek nutzbar sein. Die Zuverlässigkeit von TCP wird immer dann benötigt, wenn der korrekte Empfang der Nachrichten wichtiger ist als die Geschwindigkeit mit der sie zugestellt werden. Zu dieser Nachrichtenklasse gehören z.B. die Steuernachrichten des Kontrollzentrums oder Daten, die differentiell übertragen werden und damit von allen vorangehenden Daten abhängen. Die differentielle Datenübertragung ist immer dann unumgänglich, wenn eine Übertragung der gesamten Daten zu langsam wäre. Ein Beispiel hierfür ist das Baggern mit dem Baggersimulator. Die Änderungen am Terrain müssen für den Benutzer sofort sichtbar sein, ein hochaufgelöstes Mesh umfasst jedoch mehrere Megabyte, so dass selbst eine Gigabit-Ethernet-Verbindung schnell an ihre Grenzen stößt, wenn entsprechend viele Simulationsschritte pro Sekunde durchgeführt werden sollen. Obwohl die Zuverlässigkeit bei dieser Nachrichtenklasse wichtiger ist als die Laufzeit, sollte die Implementierung natürlich trotzdem die bestmögliche Performanz bieten. So wären z.B. durch konfigurierbare Socket-Puffergrößen mit wenig Aufwand erhebliche Leistungssteigerungen bei der Kommunikation über ein Wide Area Network (WAN) möglich.

UDP hingegen ist immer dann sehr gut geeignet, wenn der Verlust einzelner Nachrichten tolerierbar ist,

die Zustellung aber so schnell wie möglich ablaufen soll. Nicht tolerierbar sind dagegen falsch interpretierte Nachrichten oder Ausnahmen, die einen Programmabsturz zur Folge haben. Um dies zu vermeiden, ist es wenig sinnvoll, alle Mechanismen von TCP nachzubilden, denn oberhalb der Transportschicht wird man niemals die Effizienz von TCP erreichen können. Trotzdem sollte es eine Implementierung auf Basis von UDP geben, mit der zumindest schwere Fehler vermieden werden können.

Da UDP nicht wie TCP verbindungsorientiert, sondern verbindungslos arbeitet, jedes Paket also unabhängig von anderen Paketen zugestellt wird, ist es möglich, ein Paket gleichzeitig an mehrere Empfänger zu schicken. Die Netzwerkbibliothek soll es ermöglichen, dass Broadcast-Clients auch Anfragen an Broadcast-Server stellen können, bei denen die schnellste Antwort als Rückgabewert genommen wird. Dadurch kann es beim Broadcasting sogar noch leichter zu Problemen durch volle Empfangspuffer kommen. Jeder NetworkPeer der Broadcast-Nachrichten empfangen soll, muss zunächst alle Nachrichten, die er auf dem ihm zugewiesenen Port empfängt, entgegennehmen, auswerten und an die nächsthöhere Schicht weiterleiten. Versendet ein Broadcast-Client Anfragen mit Rückgabewert an mehrere Broadcast-Server und antworten diese ebenfalls über Broadcast-Nachrichten, empfängt jeder Server jede Antwort und muss sie verarbeiten, ohne dass er an der Nachricht interessiert wäre. Dadurch füllen sich zum einen die Puffer, zum anderen wird Rechenzeit verbraucht, in der die Puffer geleert werden könnten. Vermieden werden kann dieser Effekt durch eine Implementierung, bei der ein Broadcast-Server die Antworten nicht selbst wieder an alle, sondern ausschließlich an den Client sendet.

3.2.2 Nachrichtenformat

Die Hauptaufgabe der Nachrichtenklassen ist es, Nachrichten so zu serialisieren, dass sie über das Netzwerk übertragen und auf der Empfängerseite wieder deserialisiert werden können. Auf Senderseite werden Header und Payload (Nutzdaten) auf ein Byte-Array abgebildet, aus dem auf Empfängerseite die Nachricht rekonstruiert wird.

Den Aufbau von Odonet-Nachrichten zeigt Grafik 3.4. Der Header besteht aus je einem Integer-Wert für Länge, Typ und Laufnummer der Nachricht. Grundsätzlich hätten an Stelle von Integer-Werten auch Short-Werte ausgereicht, Short ist jedoch ein plattformabhängiger Datentyp (3.1.1). Den Wertebereich auf positive Werte einzuschränken (`unsigned int`) ist nicht nötig, weil die Länge einer Nachricht in der Praxis durch den verfügbaren Hauptspeicher beschränkt ist.

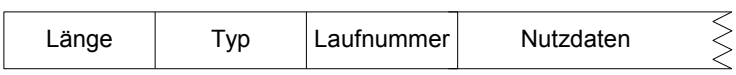


Abbildung 3.4: Nachrichtenformat

Da die Länge auf Empfängerseite zunächst unbekannt ist, muss sie mitgesendet werden. Der Typ bestimmt die Nachrichtenklasse. Neben der einfachen Nachrichtenklasse (`Message`) gibt es noch zwei weitere Klassen: die Klasse `ReturnMessage` für Antwortnachrichten und die Klasse `Autoconnect Message` für den automatischen Verbindungsaufbau. Die Klasse `LargePacketMessage`, die in der ursprünglichen Implementierung verwendet wurde um den Timeout-Wert für eine einzelne, besonders große Nachricht zu erhöhen, ist weggefallen, da bei der neuen Implementierung Timeouts pro Stub und pro Funktion von außen konfigurierbar sind. Die Laufnummer wird benötigt, um Antwortnachrichten den zugehörigen Anfragenachrichten zuordnen zu können. Die Nutzdaten werden von den spezialisierten Funktionen der Stubs geschrieben. Die Identifikationsnummern, mit denen Nachrichten und Funktionen aufeinander abgebildet werden können, sind anwendungsspezifisch und werden deshalb auch im Nutzdatenteil der Nachricht übertragen.

Ungewöhlich ist, dass die Nachrichten auf Byte-Arrays in Little-Endian-Byte-Order (höchstwertiges Byte zuerst) abgebildet werden. Meistens wird für die Übertragung über ein Netzwerk die Big-Endian-Byte-Order (höchstwertiges Byte zuletzt) verwendet, die deshalb auch als Network-Byte-Order bezeichnet wird. Da im Umfeld des Baggersimulators weitgehend PCs verwendet werden und PCs immer Little-Endian-Maschinen sind, ist es naheliegend, die überflüssige Konvertierung von Host-Byte-Order in Network-Byte-Order und zurück nicht durchzuführen. Statt dessen müssen Rechner, die mit Big-Endian-Byte-Order arbeiten, die Bytes vor dem Senden und nach dem Empfangen entsprechend drehen. Auch für die Implementierung in C# hat sich dieses Vorgehen als praktisch erwiesen, weil die zur Speicherung der Byte-Repräsentation verwendeten Klassen intern auch immer mit Little-Endian-Byte-Order arbeiten.

3.3 Kommunikationssteuerung

Um die anwendungsspezifischen Bibliotheken, und damit auch die XSLT-Stylesheets, von der verwendeten Transportschicht unabhängig zu machen, ist es nötig, die Implementierungen der Stubs und der NetworkPeers durch Indirektion zu entkoppeln. Diese Aufgabe soll die Klasse `Coordinator` übernehmen. Ein Coordinator (Abbildung 3.5) muss für jeden Stub die passenden NetworkPeers erzeugen, Verbindungen auf- und abbauen und die Aufrufe der spezialisierten Stubfunktionen auf Sende- und Empfangsoperationen der NetworkPeers abbilden können.

Je nach übergebenem `INetworkConfigEntry` muss der Coordinator eine Instanz eines Network-Servers oder NetworkClients erzeugen und bei dieser Eventhandler für ankommende Nachrichten und Verbindungsereignisse registrieren. Die Stubs wiederum registrieren Eventhandler für die selben Ereig-

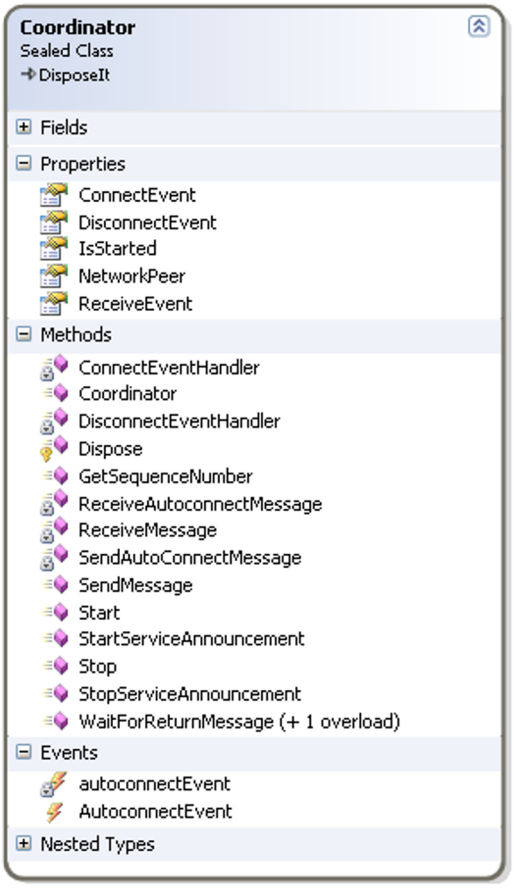


Abbildung 3.5: C#-Klassendiagramm des Coordinators

nisse bei ihrem Coordinator. Wird der Coordinator gestartet, werden entweder die Verbindungen zu den Servern aus der Konfiguration hergestellt oder der Serverdienst bereitgestellt. Sind die Stubs so konfiguriert, dass die Verbindung automatisch aufgebaut werden soll, sendet der Coordinator des ClientStubs in regelmäßigen Abständen eine `AutoConnectMessage` über einen zusätzlichen Broadcast-Client. Die Coordinatoren der ServerStubs haben dann jeweils einen zusätzlichen Broadcast-Server, der diese Dienstankündigungen empfängt und an den Coordinator weiterleitet, welcher dann eine Verbindung zum angekündigten Dienst herstellt.

Um dem Benutzer der Netzwerkbibliothek blockierende Funktionsaufrufe zu ermöglichen, muss ein Coordinator Bedingungsvariablen (condition variables) einsetzen. Der Aufruf einer ServerStub-Funktion erzeugt eine Nachricht, die der Coordinator zum Senden an den NetworkClient übergibt. Wird ein Rückgabewert erwartet, blockiert der aufrufende Thread an einer Sperre bis signalisiert wird, dass die Antwortnachricht eingetroffen ist oder bis der Timeout-Wert überschritten wurde. Trifft die Antwortnachricht ein, wird sie vom ServerStub ausgewertet und der Funktionsaufruf terminiert mit dem Rückgabewert. Im Falle eines Timeouts wird der in der Schnittstellendefinition festgelegte Fehlerwert zurückgegeben. Auf Serverseite reicht der Coordinator die Nachrichten lediglich an den ClientStub weiter, der dann

den passenden Eventhandler aufruft und dessen Rückgabewert in eine Antwortnachricht verpackt, die wiederum vom Coordinator zurückgesendet wird. Die Zuordnung, welche Antwort zu welcher Nachricht gehört, geschieht mit Hilfe von Laufnummern: Eine Antwortnachricht hat stets die selbe Laufnummer wie die Nachricht der ursprünglichen Anfrage.

Wünschenswert wäre es auch, dass mehrere Threads gleichzeitig Stubfunktionen aufrufen können, um so die Antwortzeiten zu verkürzen. Der Coordinator eines Clients müsste dann eine synchronisierte Warteschlange für die Antwortnachrichten verwalten und alle Funktionen müssten threadsicher implementiert werden. Wie in 2.4 schon festgestellt, muss die Abarbeitung der Anfragen auf Serverseite aber sehr schnell geschehen, so dass auch dafür mehrere Threads verwendet werden müssten. Weil dadurch aber an vielen Stellen Betriebssystemsperren (Mutexe oder Locks) eingesetzt werden müssten[4] und die Möglichkeiten zur Parallelisierung der Kommunkikation über ein einzelnes Socket ohnehin eingeschränkt sind, erscheint es sinnvoller, gleich mehrere Stubs an Stelle mehrerer Threads zu verwenden. Dadurch wird die Konfiguration zwar aufwändiger und es wird auch mehr Speicher belegt, die Leistungsfähigkeit dürfte bei dieser Lösung jedoch deutlich besser sein.

[4] Der Einsatz von Sperren ist teuer, weil dafür zwischen Kernel- und User-Space gewechselt werden muss.

4 Implementierung

Bei der Umsetzung des Entwurfs sollte berücksichtigt werden, dass der Programmcode leichter zu lesen und zu warten ist, wenn die Implementierungen für C# und C++ möglichst ähnlich gestaltet werden. Weil die C++-Version der Netzwerkbibliothek aber ohne eine zusätzliche Laufzeitumgebung (Common Language Runtime bzw. Mono Runtime) einsetzbar sein soll, kann nicht auf die umfangreiche Klassenbibliothek des .NET-Frameworks zurückgegriffen werden. Statt dessen muss die Implementierung in C++ dem Standard folgen und nur Bibliotheken verwenden, die für beide Betriebssysteme verfügbar sind. Dabei ist zu beachten, dass das Lizensierungsmodell auch eine kommerzielle Nutzung gestattet, aber nicht die Offenlegung des Quelltextes verlangt. Beide Forderungen werden z.B. von der BSD-Lizenz [bsd] und der LGPL [lgp] erfüllt, wobei die LGPL aber vorschreibt, dass Veränderungen an der durch sie geschützten Bibliothek publiziert werden müssen.

Beim Einsatz von Low-Level-Netzwerkbibliotheken ist zudem zu beachten, dass sie für C++ und C# zur Verfügung stehen sollten. Bei der von der ersten Implementierung der Netzwerkbibliothek verwendeten HawkNL (Hawk Network Library [haw]) ist dies nicht der Fall. Prinzipiell könnte man diese Open-Source-Netzwerkbibliothek zwar trotzdem nutzen, ihren ursprünglichen Zweck, die Programmierung zu vereinfachen, erfüllt sie dann jedoch nur noch bedingt. Bestimmte Mechanismen, wie z.B. der Aufbau einer Pseudoverbindung zur Kommunikation über UDP oder die Serialisierung von Nachrichten zu Byte-Arrays, müssten in C# mehr oder weniger aufwändig nachimplementiert werden. Bei der ersten Version der Netzwerkbibliothek wurde die HawkNL hauptsächlich verwendet, weil sie eine einheitliche Programmierschnittstelle zu den Sockets unter Windows und Linux bereitstellt. Auf Grund zahlreicher Einschränkungen und der fehlenden Unterstützung für C# wird diese Bibliothek bei der Neuimplementierung nicht mehr verwendet. Weil bei der Verwendung externer Bibliotheken aber generell das Problem besteht, dass bestimmte, möglicherweise dringend benötigte, Funktionen nicht zur Verfügung stehen und Anpassungen an die eigenen Anforderungen meist mit sehr hohem Aufwand verbunden sind, wird bei der Neuimplementierung direkt mit Sockets gearbeitet.

Wie die Grundstruktur des Entwurfs umgesetzt wurde und wie sich die Implementierung in C++ von der in C# unterscheidet, wird in Abschnitt 4.1 erläutert. Alle weiteren Abschnitte dieses Kapitels beschäftigen sich mit den Aspekten der Implementierung, die einer zusätzlichen Dokumentation bedürfen, weil

sich wichtige Zusammenhänge nicht unmittelbar aus dem Quellcode ablesen lassen.

4.1 Überblick und Unterschiede der Implementierungen in C++ und C#

Wie schon in 3.3 beschrieben, ist der `Coordinator` die zentrale Klasse der Netzwerkkommunikationsschicht. Jeder Stub hat seinen eigenen Coordinator, den er zum Senden von Nachrichten, zum Empfangen von Antworten oder für den Aufruf von Callback-Funktion benutzen kann. Handelt es sich dabei um einen Client (ServerStub), werden die `SendMessage`-Aufrufe auf Aufrufe der `SendMessage`-Methode des zugehörigen NetworkClient abgebildet, welcher die Nachricht an alle verbundenen Server überträgt. Falls eine Antwort erwartet wird, muss ein ServerStub unmittelbar nach dem Aufruf der `SendMessage`-Methode die `WaitForReturnMessage`-Methode des Coordinators aufrufen, welche die Anwortnachricht oder, im Falle eines Timeouts, `null` zurückgibt. Damit dieser Aufruf so lange blockiert bis entweder die Antwort eingetroffen ist oder das Timeout-Intervall überschritten wurde, wird beim Aufruf der `SendMessage`-Methode eine Bedingungsvariable in den Zustand „nicht signalisiert“ (nonsignaled) versetzt. Innerhalb der `WaitForReturnMessage`-Methode wird der aufrufende Thread an dieser Bedingungsvariable so lange schlafen gelegt, bis ein Empfangsthread signalisiert, dass die Nachricht eingetroffen ist oder die maximale Wartezeit überschritten wurde. Das Sperren der Bedingungsvariable muss vor dem Senden der Nachricht geschehen, damit es nicht passieren kann, dass die Antwortnachricht eintrifft, bevor `WaitForReturnMessage` aufgerufen wurde und die Bedingungsvariable auf „signalisiert“ (signaled) gesetzt wird, ohne dass auf das Signal gewartet wird[1]. In C# gibt es die einfach zu verwendende Klasse `ManualResetEvent`, die den Zugriff auf die verwendete Bedingungsvariable kapselt und synchronisiert. Für die Implementierung in C++ wurde dafür die verbreitete C-Bibliothek SDL (Simple DirectMedia Layer [sdl]) verwendet[2]. Hier muss der Zugriff auf die Bedingungsvariable allerdings manuell über einen Mutex synchronisiert werden. Dazu wurde das einfache Struct `WaitHandle` definiert, welches neben der Referenzen auf den Mutex und die Bedingungsvariable die er schützen soll, das Flag `isLocked` enthält. Dieses Flag ist notwendig, damit ein Empfangsthread beim ausführen der `ReceiveMessage`-Methode des Coordinators erkennen kann, ob noch auf die Antwortnachricht gewartet wird oder ob ein anderer Empfangsthread bereits den Eingang der Antwort gemeldet hat. Ist dies nicht der Fall, muss die Nachricht gelöscht werden. Ein weiterer Unterschied der SDL-Bedingungsvariablen zum `ManualResetEvent` von .NET ist, dass der assoziierte

[1] Das Signal ginge dann verloren, was je nach dem wie den Threads die Rechenzeit zugeteilt wird, bei lokaler Kommunikation durchaus vorkommt.

[2] Alternativ hätte genauso gut die Pthreads-Bibliothek verwendet werden können.

Mutex nach dem Eintreffen eines Signals sofort wieder gesperrt wird[3]. Damit Empfangsthreads auch nach dem Eintreffen der ersten Antwort nicht für immer blockieren, muss der Mutex nach dem bedingten Warteaufruf (`SDL_CondWaitTimeout`) wieder manuell entsperrt werden.

Ein Server (ClientStub) hingegen kann keine Anfragen versenden und wird deshalb auch niemals auf Antwortnachrichten warten. Statt dessen wird vom Coordinator die registrierte Callback-Funktion, die Methode `ProcessIncomingMessage` des zugehörigen ClientStubs, aufgerufen sobald eine Anfrage eintrifft[4]. Dort wird die Nachricht interpretiert und wiederum in einen Aufruf einer Callback-Funktion umgesetzt. Welche Funktion aufgerufen werden soll, muss der Anwender nach der Instanziierung des ClientStubs festlegen. Falls diese Funktion einen Wert zurückliefert, also nicht vom Typ `void` ist, wird eine `ReturnMessage` erzeugt, die ebenfalls der `SendMessage`-Methode des Coordinators übergeben wird. Dieser bildet den Aufruf in diesem Fall aber auf einen Aufruf der `SendReturnMessage`-Methode ab, bei dem zusätzlich zur zu sendenden Antwort das einzige Ziel der Nachricht übergeben wird[5]. Zur Umsetzung der Callback-Mechanismen wurden in C# Delegates verwendet, die die einfache Definition typsicherer Funktionszeiger erlauben. Bei der Implementierung in C++ sollten C-Funktionszeiger vermieden werden. Die Umsetzung mit typsicheren C++-Funktionszeigern war aber nicht möglich, weil die Klasse, die die Callback-Funktion enthalten soll, zum Zeitpunkt der Generierung der angepassten Bibliotheken nicht bekannt ist. Deshalb wurde die Bibliothek libsigc++ [lib] verwendet, mit der typsichere Funktionszeiger auf eine einfache Art und Weise realisiert werden können. Angelehnt an die Syntax dieser Bibliothek wird bei der automatischen Generierung der spezialisierten Stubs für jede benötigte Callback-Funktion eine Methode der Form `Connect<`*Name der Funktion*`>Slot(<`*Name der Funktion*`>Slot slot)` erzeugt, die den übergebenen `sigc::slot` mit dem zugehörigen `sigc::signal` verbindet. Der zu übergebende Slot kann vom Anwender durch einen Aufruf der Funktion `sigc::mem_fun` erzeugt werden, der die ausführende Instanz und ein Zeiger auf eine Methode dieser Instanz übergeben werden muss.

Um die Implementierung der verschiedenen NetworkPeers (siehe 4.2, 4.3 und 4.4) in C++ zu vereinfachen und möglichst nahe an der C#-Implementierung zu bleiben, wurden manche .NET-Klassen in C++ nachimplementiert. Die C++-Versionen dieser Klassen (`MemoryStream`, `BinaryReader`, `BinaryWriter` und `Bitconverter`) sind jedoch nicht vollständig, weil nur die wirklich benötigten Funktionen umgesetzt wurden um den zusätzlichen Aufwand möglichst gering zu halten. Weiterhin wurde die Klasse `Socket` definiert, die als Hüllklasse für den Zugriff auf die vom Betriebssystem be-

[3] Dieses Verhalten entspricht dem `AutoResetEvent` von .NET.

[4] `ProcessIncomingMessage` wird auch auf Clientseite aufgerufen. Dort spielt das aber nur eine Rolle, wenn Nachrichten durch ein `InterceptMessageDelegate` abgefangen werden sollen.

[5] Beim Empfang einer Nachricht wird der Sender, und damit das Ziel der Antwortnachricht, im Feld `ReturnMessageDestination` der Nachricht gespeichert.

reitgestellten Sockets fungiert und so die Unterschiede bei der Socketprogrammierung unter Windows und Linux maskiert[6].

Ein weiterer Unterschied bei den Implementierungen in den unterschiedlichen Programmiersprachen ist, dass alle C++-Klassen, die zur Konfiguration verwendet werden sollen (z.B. `NetworkConfigEntry`), auf Grund fehlender Reflection-Fähigkeiten von der Klasse `lore::Serializable` abgeleitet werden müssen. Dabei ist es erforderlich, die Methoden `LoadFrom(lore::ConfigObject*)` und `SaveTo(lore::ConfigObject*)` geeignet zu überschreiben.

Bei der Implementierung aller NetworkPeers und PeerHandles wurden ausschließlich blockierende (blocking) Sockets verwendet. Die Kommunikation läuft dabei synchron ab, was bedeutet, dass der Aufruf einer Socketfunktion erst zurückkehrt, wenn die Operation abgeschlossen ist. Im Gegensatz dazu findet bei nicht blockierenden (non-blocking) Sockets die Kommunikation asynchron statt: Alle Funktionsaufrufe kehren ohne Rückgabewert unmittelbar zurück. Sobald die Operation abgeschlossen ist und ein Ergebnis vorliegt, wird dies über ein Ereignis signalisiert und die entsprechende Callback-Funktion aufgerufen [blo]. Blockierende Sockets haben zu Unrecht einen schlechten Ruf, der wohl noch aus Windows 3.x-Zeiten herrührt [How]. Für die meisten Anwendungen sind blockierende Sockets aber das Mittel der Wahl, weil die Programmierung mit Threads durch die sequentielle Abarbeitung der Socketaufrufe erheblich vereinfacht wird[7].

4.2 TCP

Die einfachste Implementierung der Schnittstelle zur Transportschicht stellen die Klassen `TcpClient`, `TcpServer` und `TcpPeerHandle` dar. Deshalb sollen auch die gemeinsamen Grundzüge der Funktionsweise aller Implementierungen an dieser Stelle erläutert werden. Wie bei den Implementierungen für UDP oder Broadcast, instanziiert der Coordinator in seinem Konstruktor je nach Konfiguration einen Client oder einen Server und registriert bei diesem eine Callback-Funktion für ankommende Nachrichten (`ReceiveMessage`). Bei verbindungsorientierten NetworkPeers werden außerdem Callback-Funktionen für Verbindungsereignisse registriert. Damit die NetworkPeers ihre Arbeit aufnehmen, muss die `Start`-Methode des Coordinators aufgerufen werden, die dann entweder den Serverdienst startet oder die Verbindung zu allen in der Konfiguration angegebenen Servern aufbaut. Alternativ zur statischen Konfiguration der Verbindungen kann in der Konfigurationsdatei auch das Flag `autoconnect`

[6] Für die dazu notwendige bedingte Komplilierung müssen die Präprozessordefinitionen `WIN32` bzw. `LINUX` verwendet werden.

[7] Bei Anwendungen mit sehr vielen, schnell wechselnden Clients, beispielsweise bei einem Webserver, kann die Verwendung asynchroner Sockets vorteilhaft sein.

gesetzt werden, das einen ClientStub dazu veranlasst, seinen Dienst über einen periodischen Broadcast auf dem `autoconnectPort` anzukündigen. Alle ServerStubs die genauso konfiguriert sind, verbinden sich dann nach ihrem Start automatisch mit diesem ClientStub.

Der Dienst eines `TcpNetworkServers` wird wie bei allen NetworkServern immer mit der `DeployService`-Methode gestartet. Darin wird eine Liste für die Verbindungen angelegt und ein Socket („Accept-Socket“) erstellt, der an dem Netzwerkinterface, das in der Konfiguration angegeben wurde, auf Verbindungsversuche der Clients warten soll. Wird kein lokaler Name angegeben, oder wird das angegebene Netzwerkinterface nicht gefunden, wird der Socket an das erste Netzwerkinterface gebunden. Die Annahme neuer Verbindungen geschieht mit Hilfe eines dafür erzeugten Threads, der die `AcceptThread`-Methode ausführt. Dabei blockiert dieser Thread innerhalb einer Schleife an der `Accept`-Methode des Accept-Sockets bis eine neue Verbindung aufgebaut wurde. Ist der Aufruf von `Accept()` erfolgreich, liefert er einen neuen Socket zurück, der mit dem Client verbunden ist. Damit von diesem Socket Daten empfangen werden können, wird ein neuer Thread erzeugt, der dafür zuständig ist, an diesem Socket auf ankommende Daten zu warten („Receive-Thread“). Für den späteren Zugriff auf den Socket und den Thread werden sie einem neuen `TcpPeerHandle` zugewiesen, welches somit als Repräsentant für den neuen Client angesehen werden kann. Schließlich wird dieses PeerHandle zur Liste `clients` des `TcpNetworkServers` hinzugefügt und die Schleife beginnt von Neuem.

Der Verbindungsaufbau eines `TcpNetworkClient` ist weniger aufwändig: Innerhalb der `ConnectToServer`-Methode wird ein neuer Socket erstellt, der wie in der `DeployService`-Methode an ein bestimmtes Netzwerkinterface gebunden wird. Danach wird mit dem Aufruf der `Connect`-Methode des Sockets die Verbindung zum angegebenen Server hergestellt. Wie beim Server werden der verbundene Socket und ein neuer Receive-Thread anschließend einem `TcpPeerHandle` zugewiesen, welches beim `TcpNetworkClient` in die Liste `servers` aufgenommen wird.

Nachdem eine Verbindung aufgebaut wurde, kann die `SendMessageToServers`-Methode des Clients dazu verwendet werden, eine Nachricht nacheinander an alle Server zu schicken. Dazu wird die Nachricht in ein Byte-Array serialisiert, welches für jeden Server mit einem einzigen `Send`-Aufruf an den jeweiligen Socket übergeben wird[8]. Die `SendMessageToClient`-Methode eines `TcpNetworkServer` hingegen sendet die Nachricht nur an den Client, der durch das beim Aufruf übergebene PeerHandle repräsentiert wird. Dabei ist zu beachten, dass die Implementierung der `ReceiveThread`-Methode des entfernten PeerHandles bei allen Transporttypen an die Implementierungen der Sendefunktionen angepasst sein muss. Bei der Kommunikation über TCP, bei der ein Byte-Strom über einen

[8] Der Aufruf von Socketfunktionen erfordert System-Calls und ist deshalb teuer, weshalb deren Anzahl minimiert werden sollte.

zuverlässigen Kanal ausgetauscht wird, ist die Implementierung jedoch wesentlich einfacher als bei den Datagramm-orientierten Transporttypen (4.3 und 4.4): Von einem TCP-Socket können beliebig viele Bytes gelesen werden, unabhängig davon wie viele Bytes gesendet wurden. Dagegen müssen bei UDP-Sockets immer genau so viele Bytes gelesen werden, wie vorher auch gesendet wurden.

Ein `TcpPeerHandle` liest immer so viele Bytes wie möglich aus dem Puffer des assoziierten Sockets. Danach wird die Nachrichtenlänge bestimmt und unterschieden, ob für die Zusammensetzung der Nachricht alle Bytes gelesen wurden, ob noch Bytes fehlen oder ob mehr Bytes als nötig gelesen wurden. Die Bytes, die zur aktuell verarbeiteten Nachricht gehören werden in den `messageBuffer` kopiert, überschüssige Bytes landen im `overflowBuffer`. Falls nach dem ersten Aufruf der `Receive`-Funktion des Sockets noch Bytes fehlen, werden in einem weiteren Aufruf genau die Anzahl fehlender Bytes vom Socket gelesen und an den `messageBuffer` angehängt. Wurde eine Nachricht vollständig empfangen, wird sie deserialisiert und ein Empfangsereignis ausgelöst, bei dem sie der `ReceiveMessage`-Methode des Coordinators übergeben wird. Danach beginnt der Prozess wieder von neuem, wobei aber zunächst der `overflowBuffer` geleert wird indem die Länge der nächsten Nachricht ausgewertet wird und dann aus den bereits empfangenen Bytes so lange Nachricht rekonstruiert und weitergeleitet werden bis neue Daten geladen werden müssen. Diese Implementierung ist effizienter als die der alten Netzwerkbibliothek, bei der auch bei der TCP-Übertragung Pakete fester Länge verschickt wurden. Trotzdem besteht hier noch Optimierungspotenzial, wobei der Programmcode aber schon in der eben beschriebenen Version, trotz ausführlicher Kommentare, schwer zu verstehen ist.

4.3 UDP

Weil die Kommunikationsbeziehungen zwischen den verschiedenen Modulen normalerweise von längerer Dauer sind, also nicht auf wenige Aufrufe beschränkt sind, bot es sich an, auch die UDP-Implementierung verbindungsorientiert zu gestalten. Da es für Sockets zur Datagramm-Kommunikation aber keine `Accept`-Methode gibt, musste ihr Verhalten nachgeahmt werden. Ein UdpNetworkServer[9] wartet deshalb auf ein Paket eines beliebigen UdpNetworkClient, das ein bestimmtes Byte-Muster (`CONNECTION_REQUEST_PATTERN`) enthält. Trifft es ein, erstellt er einen neuen Socket und bindet ihn, damit dieser vom Betriebssystem einen neuen Port zugewiesen bekommt. Anschließend wird über den Accept-Socket[10] das `CONNECTION_REPLY_PATTERN` zusammen mit dem neuen Port an den Client gesendet

[9] Diese Bezeichnung steht verallgemeinernd für `SimpleUdpNetworkServer` und `RobustUdpNetworkServer`, entspricht aber keiner Klasse der Implementierung und wird deshalb, wie die folgenden verallgemeinernden Bezeichnungen auch, nicht gesperrt gesetzt.

[10] Würde die Verbindungsbestätigung über den neu erzeugten Socket gesendet, könnte das zu Problemen mit Firewalls führen.

und ein entsprechendes PeerHandle mit Receive-Thread zur `clients`-Liste hinzugefügt. Im Gegensatz dazu erstellt ein UdpNetworkClient einen neuen Socket, sendet das `CONNECTION_REQUEST_PATTERN` und wartet auf das `CONNECTION_REPLY_PATTERN` und den neuen Port. Trifft diese Benachrichtigung ein, wird der auf Serverseite neu angelegte Socket mittels eines `Connect`-Aufrufs als Default-Ziel für alle folgenden `Send`-Operationen festgelegt. Danach wird ein zugehöriges PeerHandle erzeugt und in die `servers`-Liste aufgenommen. Weil die Mechanismen zum Verbindungsabbau aber ebenfalls nicht fehlen dürfen, wurde festgelegt, dass ein leeres Paket als Indikator für einen kontrollierten Verbindungsabbau an den entfernten Kommunikationspartner gesendet werden soll, so dass dieser darauf entsprechend reagieren kann.

Für die Kommunikation über Datagramme gibt es zwei verschiedene Implementierungen: SimpleUdp, bestehend aus `SimpleUdpNetworkServer`, `SimpleUdpNetworkClient` und `SimpleUdp PeerHandle` und RobustUdp, bestehend aus `RobustUdpNetworkServer`, `RobustUdpNetwork Client` und `RobustUdpPeerHandle`. SimpleUdp ist die effizienteste Implementierung wenn ausschließlich kurze Nachrichten versendet werden sollen und der Sender nicht schneller ist als der Empfänger. Die Puffer der Sockets müssen dabei in jedem Fall mindestens so groß sein wie die eingestellte Paketgröße, sonst kommt es unmittelbar zu einem Socketfehler. Da in dieser einfachen Implementierung immer Pakete fester Länge versendet werden, kann die Übertragungsleistung optimiert werden, indem die Konfiguration der beteiligten Stubs angepasst wird. Wie groß die Nachrichten zur Übertragung mittels SimpleUdp sein dürfen, lässt sich nicht allgemein festlegen. Wenn eine Nachricht größer ist als die eingestellte Paketgröße, wird sie auf mehrere Pakete aufgeteilt, die nacheinander zugestellt werden. Mit steigender Senderate erhöht sich jedoch auch die Wahrscheinlichkeit eines Pufferüberlaufs, welcher unweigerlich Paketverluste zur Folge hat. Wie in 3.2.1 schon beschrieben, reichen die Konsequenzen eines Paketverlusts von der Zeitüberschreitung bis hin zum Absturz der Anwendung.

Um diese Probleme zu entschärfen, ohne eine vollständige Fluss- und Überlastkontrolle wie bei TCP zu implementieren, werden den Paketen bei RobustUdp zusätzliche Informationen hinzugefügt, wobei die Pakete diesmal von variabler Größe sind, um den zusätzlichen Overhead zum Teil zu kompensieren. Die Idee hinter RobustUdp ist, Paketverluste zu erkennen und die betroffenen Nachrichten zu verwerfen. Dazu werden den Paketen beim Senden fortlaufende Nummern gegeben, die auf der Empfängerseite überprüft werden. Das letzte Paket einer Nachricht wird mit der Laufnummer -1 als solches markiert. Bei Nachrichten, die in ein einziges Paket passen würden, müssen allerdings trotzdem zwei Pakete gesendet werden: das erste Paket, welches die Nachricht enhält und ein zweites, leeres Paket mit der Laufnummer -1, das das Ende der Nachricht markiert. Würde schon das erste Paket mit der Laufnummer -1 gesendet, könnte es passieren, dass es auf Empfängerseite falsch interpretiert wird: Wenn das abschließende Paket der vorherigen Nachricht auf Grund voller Empfangspuffer verloren gegangen ist, wird die neue Nach-

richt als der noch fehlende Teil interpretiert, was je nach Situation zu Nachrichten mit falschem Inhalt oder Socketfehlern führen kann: Eine Nachricht mit fehlerhaften Inhalt kommt genau dann zu Stande, wenn die Länge des fehlenden Pakets mit der Länge des Pakets der neuen Nachricht übereinstimmt, andernfalls kommt es auf Grund unerwarteter Datagramm-Größen zu Socketfehlern. In den Stresstests (siehe 5.4) machte sich der Verlust eines einzelnen Pakets in der Regel durch zwei bis drei aufeinanderfolgende Timeouts bemerkbar, Abstürze oder Nachrichten mit falschem Inhalt wurden effektiv verhindert. Der Grund für diese Zeitüberschreitungen ist, dass der Sender nicht weiß, dass ein Paket verloren gegangen ist und deshalb ungehindert Pakete sendet, die der Empfänger so lange verwerfen muss, bis er wieder den Beginn einer neuen Nachricht empfängt. Weil der Puffer eines Sockets nach dem Verlust eines Pakets aber nicht einfach geleert werden kann, sondern alle Datagramme zunächst entgegengenommen werden müssen, kann es eine Weile dauern bis der Empfangspuffer geleert ist, so dass es passieren kann, dass der Anfang nachfolgender Nachrichten verloren geht, welche dann wiederum verworfen werden müssen.

4.4 Broadcast

Für das Broadcasting von Nachrichten gibt es ebenfalls eine einfache (SimpleBroadcast) und eine robuste Implementierung (RobustBroadcast). Da die Kommunikation beim Broadcasting nicht verbindungsorientiert erfolgt, werden keine Listen mit PeerHandles zur Repräsentation der entfernten Kommunikationspartner benötigt. Statt dessen hat jeder BroadcastServer oder BroadcastClient ein BroadcastPeerHandle, das sog. `receivePeerHandle`, über dessen Socket der assoziierte Receive-Thread Nachrichten entgegen nimmt und einen sog. `sendSocket`, über den Nachrichten versendet werden können. Die Verwendung zweier Sockets ist zwingend notwendig, wenn Antworten auf Broadcast-Anfragen nicht wieder auf dem Broadcast-Kanal gesendet werden sollen: Selektive Antwortnachrichten müssen über einen `SendTo`-Aufruf mit der Zieladresse als Parameter an den Socket übergeben werden. Anders als beim Broadcasting wird ein gewöhnliches UDP-Paket aber vom ersten Empfänger aus dem Netz genommen, so dass es passieren kann, dass das Paket vom Sender selbst empfangen und konsumiert wird und deshalb niemals beim eigentlichen Empfänger ankommt. Aus diesem Grund muss ein Kommunikationsteilnehmer auf unterschiedlichen Ports senden und empfangen, wozu wiederum zwei Sockets benötigt werden. Damit bei der Konfiguration von BroadcastPeers nicht zwei Ports angegeben werden müssen, wurde festgelegt, dass ein BroadcastServer auf dem Port aus der Konfigurationsdatei empfängt und auf dem nächsthöheren Port[11] sendet, wohingegen ein BroadcastClient auf dem Port aus der Konfiguration sendet und auf dem nächsthöheren Port empfängt.

Die Implementierung der Sende- und Empfangsfunktionen für das Broadcasting unterscheidet sich nur

[11] nächsthöherer Port = Port + 1

unwesentlich von denen für Udp, wobei im Receive-Thread bei der RobustBroadcast-Implementierung für jedes empfangene Paket zusätzlich überprüft wird, ob es von dem Sender stammt, von dem das erste Paket der aktuellen Nachricht entgegen genommen wurde. Ist dies nicht der Fall, wird das Paket verworfen, damit es nicht passieren kann, dass eine Nachricht aus den Paketen unterschiedlicher Kommunikationsteilnehmer zusammengesetzt wird.

4.5 Kontrollzentrum

Für die Kommunikation zwischen den verschiedenen Modulen und dem Kontrollzentrum wurden spezielle Stubs geschaffen, die allgemein als `ControlCentreStub`s bezeichnet werden. Davon abgeleitet gibt es, angelehnt an die Terminolgie der prototypischen Implementierung des Kontrollzentrums von Benny Wegener, die Klassen `ControllerStub` und `InstanceStub`. Ein `InstanceStub` wird auf Seite des Kontrollzentrums verwendet, um eine bestimmte Instanz einer Komponente anzusprechen. Mit einem `ControllerStub` können Komponenten Nachrichten an das Kontrollzentrum schicken.

Die erste Schwierigkeit bei der Implementierung bestand darin, dass beide Arten von Stubs proaktiv Nachrichten versenden (Clientrolle), ebenso aber auch Anfragen beantworten können müssen (Serverrolle), was die strenge Unterscheidung von Client und Server aber nicht erlaubt (siehe 3.1.2). Damit nicht bei jeder Komponente zwei Stubs für die Kommunikation mit dem Kontrollzentrum instanziiert und konfiguriert werden müssen, wurde eine Sonderlösung geschaffen, so dass jeder `ControlCentreStub` gleichzeitig als Client und als Server agieren kann. Intern arbeitet jeder `ControlCentreStub` dazu mit zwei Coordinatoren, einer mit einem `TcpNetworkServer`, einer mit einem `TcpNetworkClient`: Werden Funktionen des `ControlCentreStub` aufgerufen, sendet der Client-Coordinator die resultierenden Anfragen an den entfernten `ControlCentreStub`. Ankommende Anfragen werden vom Server-Coordinator in Aufrufe der registrierten Callback-Funktion umgesetzt.

Die zweite Anforderung die schwierig umzusetzen war, ist dass das Kontrollzentrum die physische Verteilung aller laufenden Module erfassen und jedes Modul einzeln ansprechen können muss. Eine statische Konfiguration der Verbindungen ist dabei nicht ausreichend, da weder angenommen werden kann, dass das Kontrollzentrum bereits gestartet wurde bevor die anderen Komponenten gestartet werden, noch dass alle Komponenten bereits laufen wenn das Kontrollzentrum gestartet wird. Der Mechnismus, den herkömmliche Stubs zum automatischen Verbindungsaufbau verwenden (siehe 4.2), ist aber ebenfalls ungeeignet, weil die individuelle Adressierbarkeit der Instanzen dann nicht gegeben ist. Um dieses Problem zu lösen wurde der `ControlCentreAnnouncer` geschaffen. Dieser wird vom Kontrollzentrum dazu benutzt, alle Module die einen `ControllerStub` gestartet haben aufzufinden und eine Verbin-

dung zu ihnen herzustellen. Dazu sendet der `ControlCentreAnnouncer` auf dem angegebenen Port Broadcast-Mitteilungen und generiert für jeden noch nicht registrierten `ControllerStub` von dem eine Antwort empfangen wird einen neuen, bereits verbundenen `InstanceStub`. Im einzelnen läuft der Verbindungsaufbau wie folgt ab: Der `ControllerStub` erzeugt seinen Client und seinen Server mit den Parametern aus der Konfigurationsdatei[12], wobei aber zunächst nur der Server gestartet wird. Zusätzlich dazu wird ein BroadcastClient erzeugt, der auf Nachrichten vom `ControlCentreAnnouncer` wartet. Sobald eine solche Nachricht eintrifft, antwortet der `ControllerStub` mit einer Nachricht, die die IP-Adresse und den Port seines Servers enthält. Wird diese Nachricht wiederum vom Announcer empfangen, erzeugt dieser einen neuen `InstanceStub` und speichert die entfernte IP-Adresse zusammen mit dem Port als Schlüssel in einer Hashtabelle um mehrfache Verbindungen zu vermeiden. Danach werden zuerst der Server und dann der Client des neuen `InstanceStub` gestartet. Durch den Verbindungsversuch des `InstanceStub` wird beim Server des `ControllerStub` ein Ereigniss ausgelöst, mit dem die IP-Adresse und der Port des Servers des `InstanceStub` ermittelt werden kann, zu dem unmittelbar danach die Verbindung hergestellt wird. Innerhalb dieser Ereignisbehandlungsroutine wird auch der BroadcastClient gestoppt, so dass keine weiteren Nachrichten vom `ControlCentreAnnouncer` empfangen werden können. Weil der gesamte Verbindungsaufbau aber doch etwas dauert, kann es vorkommen, dass die Anfragen des `ControlCentreAnnouncer` mehrfach beantwortet werden, weshalb dieser letzlich dafür verantwortlich sein muss, dass zu jedem `ControllerStub` nur ein `InstanceStub` erzeugt wird. Beim Abbau der Verbindungen ist zu beachten, dass ein `InstanceStub` dem `ControlCentreAnnouncer` mitteilt, dass seine Verbindung verloren ging, so dass der `ControlCentreAnnouncer` den entsprechenden Schlüssel aus der Hash-Tabelle löscht und damit wieder eine Verbindung zum korrespondierenden `ControllerStub` aufbauen kann. Verliert ein `ControllerStub` die Verbindung zu seinem `InstanceStub`, wird der BroadcastClient wieder gestartet, so dass er wieder Nachrichten vom `ControlCentreAnnouncer` empfangen kann.

Abschließend soll anhand von Listing 4.1 gezeigt werden, wie eine Minimalkonfiguration für einen `ControlCentreAnnouncer` und einen passenden `ControllerStub` aussieht.

4.6 Automatische Generierung benutzerdefinierter Bibliotheken

Als Technologie für die automatische Generierung benutzerdefinierter Stubs bietet sich XML zur Definition der gewünschten Funktionen (3.1.1), in Kombination mit XSL-Transformationen zur automatischen Codegenerierung an. Die Schnittstellenbeschreibungen lassen sich mit XML einfach erstellen und sind gut menschen- und maschinenlesbar. Viele XML-Editoren bieten außerdem Eingabehilfen an, sofern das

[12] Die Kommunikation zwischen Kontrollzentrum und Komponenten wird momentan ausschließlich über TCP unterstützt.

Listing 4.1: Konfigurationsdatei für ControlCentre-Kommunikation

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration name="controlcentre_configuration">
  <section id="network">
    <object id="ControlCentreAnnouncer" type="ControlCentreAnnouncer">
      <attr name="autoconnectport">4000</attr>
    </object>
    <object id="ControllerStubServer" type="ControllerStub">
      <attr name="port">3000</attr>
    </object>
    <object id="ControllerStubClient" type="ControllerStub">
      <attr name="autoconnectport">4000</attr>
    </object>
  </section>
</configuration>
```

zu verwendende XML-Schema bekannt ist. Ein weiterer großer Vorteil ist, dass kein spezieller Interface-Compiler wie z.B. bei CORBA benötigt wird, sondern ein Standard-XSL-Prozessor verwendet werden kann. Aus lizenzrechtlichen Gründen und um die Erweiterbarkeit zu gewährleisten, wurde von Stefan Spickereit trotzdem ein kompakter XSL-Prozessor mit dem .NET-Framework entwickelt. Dieser generiert aus der Schnittstellenbeschreibung und den Transformationsregeln, die in den XSLT-Stylesheets festgelegt sind, den entsprechenden Quellcode.

Die Implementierung der XSLT-Stylesheets, die in Kooperation mit Stefan Spickereit erfolgte, orientierte sich an den bereits vorhandenen Stylesheets für C++ von Jan Stühmer. Die neuen Datentypen (3.1) und Features erforderten jedoch eine Anpassung der Transformationsvorschriften. Hier sind, neben den Möglichkeiten zur Definition funktionsspezifischer Fehlerwerte und Timeouts, vor allem die Threadsicherheit und die Möglichkeit zum Abfangen von Nachrichten zu nennen.

Nachdem die XML-Beschreibung der Schnittstelle in C++- oder C#-Code transformiert wurde, müssen daraus noch die entsprechenden dynamischen Bibliotheken kompiliert werden.

Um den Erstellungsprozess zu vereinfachen, wurden Batch-Dateien bzw. Shell-Scripts bereitgestellt, die nur noch mit dem Namen der Schnittstelle als Parameter aufgerufen werden müssen, um die gewünschten Dynamic Link Libraries zu erhalten. Intern kommt dabei NAnt zum Einsatz, das die XSLT-Transformation und die Kompilierung ausführt. Für C# wird zusätzlich eine kompilierte Hilfedatei (.chm) generiert. Abbildung 4.1 skizziert diese Verarbeitungskette.

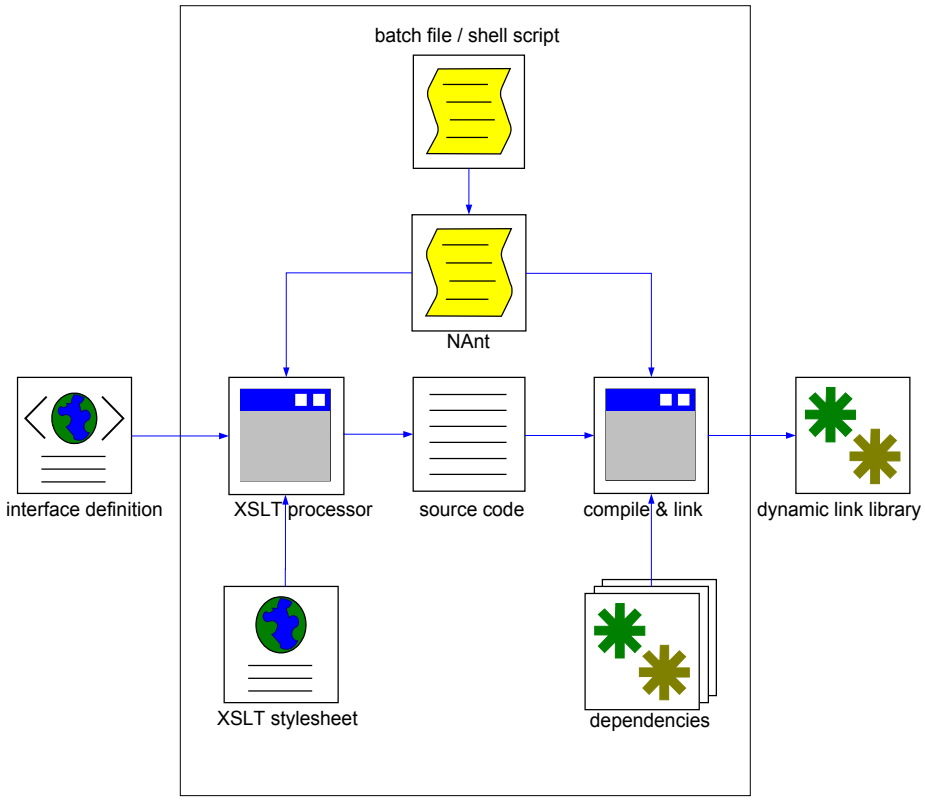


Abbildung 4.1: Transformationspipeline

5 Automatisiertes Testen

Automatisierte Tests spielen mittlerweile eine enorm wichtige Rolle bei der Qualitätssicherung von Software. Die klassische Prüfung der Fehlerfreiheit von Anwendungen durch Tester, die oft erst nach der vollständigen Implementierung stattfindet, ist angesichts der Komplexität heutiger Anwendungen und Systemumgebungen nicht mehr ausreichend.

Durchschnittlich kommen auf 1000 Zeilen Programmcode mindestens zwei Fehler [RS]. Um auch bei großen Projekten, deren Codebasis mehrere Millionen Zeilen umfassen kann, die Anzahl der Fehler klein zu halten, sind neue Strategien zur Qualitätssicherung notwendig. Neben zahlreichen Verbesserungen bei den integrierten Entwicklungsumgebungen haben sich deshalb neue Testwerkzeuge und neue Vorgehensmodelle bei der Softwareentwicklung etabliert.

Ziel der Testautomation ist es, Fehler aufzudecken und die Suche nach ihren Ursachen zu erleichtern. Dieses Ziel kann jedoch nur erreicht werden, wenn das Testen in den Entwicklungsprozess integriert ist. Der Programmierer muss also selbst zum Tester werden und die Teilergebnisse seiner Arbeit kontinuierlich überprüfen. Zunächst kostet die Testautomatisierung aber viel Zeit und Geld. Sowohl [KBP01], als auch [Rä04] veranschlagen einen im Vergleich zu einer manuellen Überprüfung zehnmal höheren Aufwand für die Erstellung automatisierter Tests. Zusätzlich müssen die Tests während der ganzen Entwicklungszeit gepflegt werden. Änderungen und Erweiterungen des Programms oder neue Szenarien, die ebenfalls geprüft werden sollen, bedingen eine Anpassung und Weiterentwicklung der Tests. Die schnelle Reproduzierbarkeit der Testabläufe kann die zusätzlichen Kosten allerdings schnell wieder relativieren. Ist die Basisfunktionalität der Software stets durch Tests abgesichert, können kritische Fehler oft frühzeitig erkannt werden. Das Risiko, in späten Projektphasen noch schwerwiegende Fehler beseitigen zu müssen, kann durch das parallele Testen minimiert werden.

Die Automatisierung von Tests kann und soll die klassischen Methoden zur Sicherstellung der Korrektheit von Software aber nicht ersetzen. Die Modellvalidierung und Code-Reviews durch das Entwicklungsteam adressieren andere Bereiche der Qualitätssicherung. Ebenso wenig sollte bei größeren Projekten auf eigenständige Tester oder eine Beta-Version für ein breites Testpublikum verzichtet werden. Außenstehende haben oft eine andere Herangehensweise und andere Erwartungen als die Programmierer, so dass von ihnen ganz andere Mängel wahrgenommen und wertvolle Verbesserungsvorschläge gemacht

werden können.

5.1 Arten von Tests

Die Popularität von automatischen Tests ist in den letzten Jahren enorm gestiegen. Das häufig diskutierte Vorgehensmodell des Extreme Programming dürfte dazu einen großen Teil beigetragen haben. Viele Praktiken des Extreme Programming, wie z.B. das Refactoring und die kontinuierliche Integration (Continous Integration), sind erst mit automatisierten Tests effizient durchführbar. Für die Testgetriebene Entwicklung (Test Driven Development) bilden sie sogar die Grundlage. Allerdings werden „Automatisiertes Testen“ und „Unit Testing“ fälschlicherweise oft synonym verwendet, weshalb dieses Kapitel Unit Tests von anderen Testarten abgrenzen soll.

5.1.1 Unit Tests

Ein Unit Test ist ein automatisierter Test, der einzelne Programmeinheiten auf funktionale Korrektheit und Vollständigkeit prüft. Wie groß diese Programmeinheiten sein dürfen ist unter Experten umstritten. Gängige Praxis ist, dass für jede zu testende Klasse eine Testklasse erstellt wird. Die traditionelle Definition von Unit Tests fordert außerdem, dass nur eine einzige Klasse in Isolation von allen anderen getestet wird [Beu]. Diese Forderung wird von den Programmierern aber meist vernachlässigt. Das Ergebnis sind Tests, die von anderen Klassen (Integrationstests, siehe 5.1.2) oder einer komplexen Infrastruktur abhängig sind (Systemtest, siehe 5.1.3).

Michael Feathers, der Entwickler von CppUnit, hat deshalb einen Satz von Regeln verfasst, der zwar keine völlige Isolation der zu testenden Klasse fordert, den Begriff „Unit Test“ aber trotzdem auf ein sinnvolles Maß einschränken soll [Fea]. Nach seiner Definition ist ein Test kein Unit Test wenn:

- er ein Datenbanksystem benutzt
- er über ein Netzwerk kommuniziert
- er das Dateisystem berührt
- er nicht gleichzeitig mit jedem anderen Unit Test durchgeführt werden kann
- wenn die Umgebung speziell konfiguriert werden muss

Letztlich geht es bei diesen Einschränkungen darum, dass solche Tests in jedem Fall sehr schnell und ohne großen Aufwand durchführbar sein müssen. Feathers verwischt dadurch aber die Grenze zwischen isolierten Unit Tests, bei denen ausschließlich die Korrektheit einer einzelnen Klasse überprüft wird, und funktionalen Tests, bei denen alle Teile des Systems involviert sein können. Anhänger des Test Driven

Development argumentieren aber, dass es zwar sinnvoll sei, schnelle und langsame Tests voneinander zu trennen, es müsse aber trotzdem zwischen Unit Tests und Integrations- oder Systemtests unterschieden werden. Durch das isolierte Testen einzelner Klassen sei es einfacher, eine hohe Testabdeckung zu erreichen, lose gekoppelten Code zu erzeugen und Änderungen schnell zu integrieren [New].

Oft kann eine Klasse ihre Aufgaben aber nicht ohne die Hilfe anderer Klassen erfüllen. Soll eine solche Klasse trotzdem unabhängig von den anderen getestet werden, können die Instanzen der Hilfsklassen durch Stubs oder Mock-Objekte ersetzt werden. Stubs sind möglichst einfache Versionen dieser Hilfsklassen. Sie implementieren zwar dieselbe Schnittstelle, ihre Funktionen liefern normalerweise aber nur vordefinierte Werte zurück. Sie werden oft eingesetzt, um Objekte zu ersetzen, die schwierig zu erzeugen oder zu manipulieren sind. Deshalb sind sie meist an den Rändern eines Systems zu finden, wo sie Datenbanken, Webservices oder andere Dienste ersetzen. Mock-Objekte haben prinzipiell dieselbe Aufgabe wie Stubs, werden aber anders benutzt. Martin Fowler, einer der Mitbegründer des Extreme Programming, sieht den wesentlichen Unterschied zwischen Stubs und Mocks im Teststil den sie vorantreiben [Fow]. Während bei den Stubs das zustandsbasierte Testen gefördert wird, wird durch Mock-Objekte ein interaktionsbasierter Teststil unterstützt. Unit Tests sind in der Praxis meist zustandsbasierte Tests: Eine Funktion eines Objekts wird aufgerufen und anschließend wird überprüft, ob der zurückgegebene Wert korrekt ist, oder ob das Objekt den gewünschten Zustand angenommen hat. Noch nicht so weit verbreitet ist der interaktionsbasierte Ansatz, bei dem der Programmierer angeben kann, welche Funktionen des Mock-Objekts von der zu testenden Klasse während des Tests aufgerufen werden müssen und welcher Wert dann zurückgegeben werden soll. Hier wird also vor allem getestet, ob bestimmte Funktionen aufgerufen wurden und nicht was ihr Aufruf bewirkt. Interaktionsbasierte Tests sind also eher Integrations- als Unit Tests, auch wenn Mock-Objekte ein wichtiges Hilfsmittel zum isolierten Testen einzelner Klassen darstellen. Ob der Aufwand betrieben werden sollte, eine bestimmte Unit in Isolation zu testen, muss von Fall zu Fall abgewägt werden. Unit Tests können Integrations- oder Systemtests nicht ersetzen, sie können aber die Zahl der notwendigen Testfälle in diesen Phasen reduzieren. Außerdem wird das Auffinden von Fehlerursachen durch das isolierte Testen von Klassen erheblich vereinfacht: Der Fehler kann nur in der getesteten Klasse lokalisiert sein.

5.1.2 Integrationstests

Die Aufgabe von Integrationstests ist es, das korrekte Zusammenspiel von Klassen oder Komponenten sicherzustellen. Normalerweise werden die Programmeinheiten schrittweise zu größeren Einheiten zusammengesetzt, um im Fehlerfall die Ursachen schneller auffinden und beheben zu können. Für die Richtung, in welcher die Assemblierung stattfinden sollte, lässt sich keine allgemein gültige Empfehlung

geben. Je nach Architektur kann sich ein Top-Down-Ansatz, also von der Benutzerschnittstelle zur Geschäftslogik und von dort zur Datenhaltung oder, in die umgekehrte Richtung, ein Bottom-Up-Ansatz als günstiger erweisen. Bei einer mehrschichtigen Architektur kann es vorteilhaft sein, sich von den inneren Schichten nach oben und unten vorzuarbeiten (Middle-Out).

Im einzelnen müssen Integrationstests die Korrektheit von Transaktionen über mehrere Programmeinheiten, das Zusammenspiel der verschiedenen Schichten und die richtige Reaktion auf Ereignisse (Events) sicherstellen. Integrationstests sollten so bald wie möglich durchgeführt werden, um Schwächen im Entwurf frühzeitig aufzudecken.

5.1.3 Systemtests

Systemtests sind die letzten Tests vor den Abnahmetests mit dem Kunden. Sie sollen sicherstellen, dass das Gesamtsystem alle Anforderungen zufriedenstellend erfüllt. Der Systemtest überprüft, ob die gewünschte Funktionalität vollständig umgesetzt ist, wie leistungsfähig das System bei hoher Belastung ist, ob Sicherheitsaspekte ausreichend berücksichtigt wurden und wie gut das System in verschiedenen Umgebungen funktioniert. Daneben betrifft der Systemtest auch implizite Anforderungen, also Anforderungen, die es nicht ins Pflichtenheft geschafft haben, vom Kunden aber erwartet werden. Neben der Benutzerfreundlichkeit muss außerdem noch das Benutzerhandbuch überprüft werden.

Systemtests stellen also im wesentlichen die Vollständigkeit der Umsetzung der Anforderungen sicher. Für einen Systemtest gelten Anforderungen aber erst dann als umgesetzt, wenn sie implementiert und getestet wurden. Damit ist es ebenfalls Gegenstand eines Systemtests, zu überprüfen, ob die Funktionstests vollständig sind. Dazu werden die Dokumente aus der Analysephase herangezogen. Aus den dort angegebenen Anwendungsfällen und Szenarien ergeben sich direkt die Anforderungen, deren korrekte Umsetzung durch Unit- und Integrationstests überprüft worden sein muss.

Soll die Software auf unterschiedlichen Betriebssystemen lauffähig sein, muss das Verhalten unter den verschieden Umgebungen untersucht werden. Dabei ist zu beachten, dass es nicht nur bei unterschiedlichen Systemfamilien wie Linux, Windows oder Mac OS zu abweichendem Laufzeitverhalten kommen kann, sondern auch innerhalb der Systemfamilien gravierende Unterschiede zwischen den Versionen existieren. Zum einen liegt das daran, dass die Systemkomponenten anders verwaltet werden, zum anderen können auch Updates oder Service-Packs das Laufzeitverhalten beeinflussen. Dazu kommen außerdem noch unterschiedliche Versionen von Komponenten, von denen die Software abhängig ist (DLL Hell). Weiterhin sollten auch die Installierbarkeit, die Konfigurierbarkeit und schließlich die Deinstallierbarkeit auf den verschiedenen Plattformen überprüft werden. Neben dem Betriebssystem können auch die Hardware und die zugehörigen Treiber einen entscheidenden Einfluss auf den korrekten Ablauf eines

Programms nehmen. Dies ist bei der Konfiguration der Testsysteme zu berücksichtigen.

Die Benutzerfreundlichkeit der Software und das zugehörige Handbuch sollten am besten durch ein unabhängiges Team getestet werden. Die Entwickler neigen oft dazu, „betriebsblind“ zu werden und fehlerhaftes oder störendes Verhalten zu übersehen.

Um auch Systemtests möglichst früh durchführen zu können, sollten schon vor dem Release noch unfertige Versionen fixiert und zum Testen freigegeben werden.

5.2 Testwerkzeuge

Die Automatisierung von Testabläufen ist ein mühsames Unterfangen, wenn geeignete Werkzeuge fehlen. Zunächst muss eine Testumgebung bereitgestellt werden. Dazu gehören Testdaten, Betriebssystemressourcen und im Kontext von Systemtests auch die Hardwareumgebung. Soll ein Test häufig wiederholt werden, kann die automatisierte Initialisierung der Testumgebung Zeit sparen und Fehler vermeiden. Um die Tests auszuführen, kann eine eigens dafür geschriebene Testapplikation, eine Script-Sprache oder ein Testframework zum Einsatz kommen. Nach der Ausführung müssen die Ergebnisse ausgewertet und protokolliert werden können. Bei Programmen die eine Datenbank verwenden, muss auch von außerhalb des zu testenden Programms darauf zugegriffen werden können. Um verteilte Systeme zu testen, benötigt man möglicherweise Werkzeuge, die den Netzwerkverkehr mitschneiden oder sogar manipulieren können. Sollen Tests automatisiert werden, die eine Anwendung über eine graphische Benutzerschnittstelle ansprechen, sind Capture-and-Replay-Werkzeuge vonnöten.

In diesem Kapitel soll eine Auswahl an Werkzeugen vorgestellt werden, die die Testautomation bei der Implementierung der Netzwerkschicht erleichtern können. Auf Werkzeuge zum Testen von Datenbankzugriffen oder graphischen Benutzerschnittstellen (GUI) wird deshalb nicht eingegangen. Obwohl eine Netzwerkbibliothek für ein verteiltes Simulations- und VR-System entwickelt wird, kann das Testen von verteilten Systemen wegen der Komplexität dieses Themas nur kurz in 5.5 angesprochen werden.

5.2.1 NUnit

NUnit [TPCF] ist sicherlich eines der populärsten Unit-Testing-Frameworks überhaupt. Ursprünglich war es eine einfache Portierung von JUnit auf die .Net-Sprachen[1]. Mit der Version 2.0 wurde es aber komplett überarbeitet, um .Net-spezifische Features wie Attributierungs- oder Reflectionmechanismen voll ausnutzen zu können. Die aktuelle Version 2.2.8 ist jedoch nur für Windows verfügbar, das letzte offizielle Release für Mono, Version 2.2, ist schon recht alt und unterstützt nicht alle Features. NUnit ist Open Source Software und wird unter der zlib/libpng-Lizenz verbreitet, die den Gebrauch nur sehr wenig einschränkt [zli].

Allen Testframeworks ist gemeinsam, dass sie dem Tester viel Arbeit abnehmen, so dass dieser sich auf das Schreiben der Testfälle konzentrieren kann. Die Testfälle werden in derselben Sprache wie die zu testende Unit verfasst und zu sogenannten Test-Suites zusammengefasst. Meistens ist es günstig, pro Klasse eine Test-Suite und pro Methode mindestens einen Testfall zu definieren. Test-Suite sollten wiederum zu Testprojekten zusammengefasst werden, welche dann, je nach Framework, zu ausführbaren Konsolenprogrammen oder dynamischen Bibliotheken kompiliert werden können. Die einfachere Variante sind die Konsolenprogramme. Sie beschränken sich normalerweise auf das Ausführen der Testfälle und eine einfache Ausgabe der Testergebnisse. Komfortabler sind Testprogramme mit einer graphischen Benutzerschnittstelle, bei denen die Tests als dynamische Bibliotheken geladen werden.

Um die Reproduzierbarkeit der Testergebnisse zu gewährleisten, muss ein Testframework dafür sorgen, dass jeder Testdurchlauf unter möglichst gleichen Voraussetzungen statt findet. Deshalb ruft der Testrunner des Frameworks vor jedem einzelnen Test die Setup-Methode der Testklasse auf, mit der weitgehend gleiche Bedingungen geschaffen werden sollen. Die Gesamtheit der Objekte, die zum Ausführen der Tests einer Testklasse benötigt wird, bezeichnet man als Test-Fixture. Damit alle Tests unabhängig voneinander und in beliebiger Reihenfolge durchgeführt werden können, sollte das Erzeugen und Initialisieren von Instanzen der zu testenden Klasse und eventuell benötigter Hilfsklassen immer in der Setup-Methode geschehen. Manchmal ist es jedoch nicht nötig oder nicht praktikabel, bestimmte Objekte vor jedem Test neu zu kreieren, etwa weil die Objekte zustandslos sind oder weil ihre Erzeugung aufwändig ist. In diesem Fall sollten sie vom Konstruktor der Testklasse oder von der von manchen Testframeworks dazu angebotenen Test-Fixture-Setup-Methode bereitgestellt werden.

Die Testfälle selbst werden ebenfalls als Methoden formuliert, die wiederum Methoden des Testobjekts aufrufen und mit Hilfe von Zusicherungen (Assertions) überprüfen, ob die zurückgegebenen Werte bzw.

[1] SUnit, das erste Unit-Testing-Framework überhaupt wurde von Kent Beck für Smalltalk entwickelt. JUnit ist eine SUnit-Portierung auf Java. Fast alle Unit-Testing-Frameworks sind einfache Übersetzungen von SUnit oder JUnit in die Zielsprache.

die Änderungen des Objektzustands den Erwartungen entsprechen. Nach jedem Test wird die Tear-Down-Methode der Testklasse aufgerufen. Sie stellt das Gegenstück zur Setup-Methode dar und soll die allokierten Ressourcen nach jedem Test wieder freigeben.

NUnit nutzt modernste Sprachmittel, um das Schreiben von Tests möglichst einfach zu gestalten. Mittels Attributen werden Testklassen und Testmethoden als solche markiert, so dass das bei den anderen Testframeworks[2] übliche Ableiten von einer Basistestklasse und das explizite Registrieren der Testmethoden entfällt (siehe 5.2.3). Wie bequem damit Tests erstellt werden können, soll an Hand eines Testfalls der `Message`-Klasse gezeigt werden (Listing 5.1).

Erstellt man daraus eine dynamische Bibliothek, kann man sie mit der Anwendung NUnit-GUI öffnen und den Test starten. Mit diesem Testrunner lassen sich mehrere Assemblies laden, wobei über Checkboxen ausgewählt werden kann, welche Tests ausgeführt werden sollen. Abbildung 5.1 zeigt das Fenster der Anwendung nach der Durchführung des Tests.

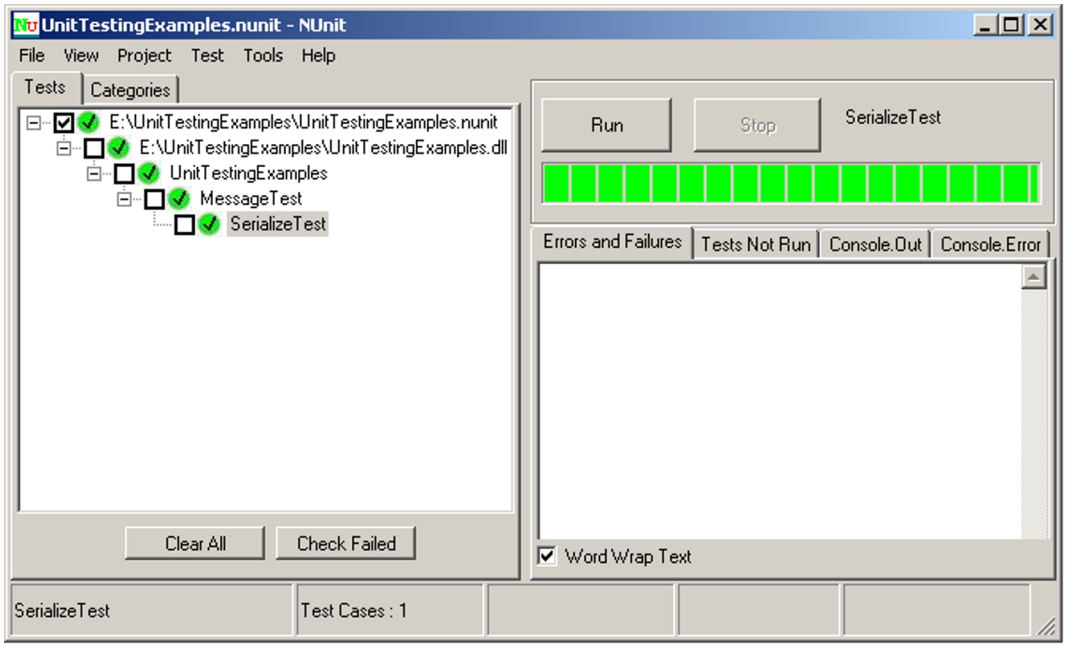


Abbildung 5.1: NUnit-GUI unter Windows

NUnit ist frei verfügbar, einfach zu benutzen und bietet in der aktuellen Version eine Vielzahl von vordefinierten Assertions und Attributen, von denen eine Auswahl in den Tabellen B.1 und B.2 zusammengefasst sind. Dass man NUnit aber trotzdem noch verbessern kann, wird der nächste Abschnitt zeigen.

[2]Seit der kürzlich erschienenen Version 4.0 unterstützt JUnit ebenfalls die Testdeklaration durch Attribute.

Listing 5.1: MessageTest.cs

```
using NUnit.Framework;
using odonet_cs;
namespace UnitTestingExamples
{
        [TestFixture]
        public class MessageTest
        {
                private Message message;
                [SetUp]
                public void Setup()
                {
                        this.message = new Message();
                }
                /// <summary>
                /// This Test determines, if a simple message can be serialized properly.
                /// </summary>
                [Test]
                public void SerializeTest()
                {
                        //"43" will be the payload of the message
                        this.message.WriteInt(43);
                        this.message.SequenceNumber = 7;
                        //4 bytes for length
                        //4 bytes for message type
                        //4 bytes for sequence number
                        //4 bytes for the payload
                        byte[] expected = {
                                16, 0, 0, 0,
                                0, 0, 0, 0,
                                7, 0, 0, 0,
                                43, 0, 0, 0
                        };
                        byte[] result = this.message.Serialize();
                        Assert.AreEqual(expected, result);
                }
        }
}
```

5.2.2 Zanebug

Zanebug [zan] basiert auf NUnit und erweitert es, ist jedoch ausschließlich für Windows verfügbar. Neben einigen neuen Attributen fallen im Vergleich zu NUnit vor allem die wesentlich umfangreichere graphische Benutzerschnittstelle und Funktionen zur Performance-Messung auf. Die aktuelle Version 1.6.0 ist größtenteils kompatibel mit NUnit in der Version 2.2.7. Zanebug ist ebenfalls Open Source Software und ist unter der Apache2.0-Lizenz verfügbar, die der zlib/libpng-Lizenz sehr ähnlich ist [apa].

Mit den Attributen TestSetUp und TestTearDown ist es möglich, testspezifische Setup- und Tear-Down-Methoden (`TestSetUp`, `TestTearDown`) anzugeben. Diese Erweiterung ist zwar nicht unverzichtbar, schließlich könnte einfach eine weitere Testklasse mit anderen Setup- und Tear-Down-Methoden definiert werden, kompakter und zeitsparender ist sie aber in jedem Fall. Ebenfalls praktisch ist das Repeat-Attribut, mit dem angegeben werden kann, wie oft ein Test durchlaufen werden soll. Bei der Alternative, einer von Hand programmierten Schleife innerhalb der Testmethode, ist zu beachten, dass die Setup- und Tear-Down-Methoden nur einmal vor und nach jedem Test vom Testrunner aufgerufen werden. Zustandsbehaftete Objekte müssen also innerhalb der Schleife erzeugt werden, um die Forderung nach den gleichen Bedingungen für jeden Testdurchlauf nicht zu verletzen. Für die Programmierung mit mehreren Threads gibt es die Möglichkeit, einen Test mit `IsMultithreaded` zu markieren. Der Testrunner erzeugt dann transparent einen neuen Thread, der den Test ausführt. Deadlocks und andere Fehler die durch Nebenläufigkeit entstehen, lassen sich wegen der Wiederholbarkeit der Tests mit einer größeren Wahrscheinlichkeit als bei manuellen Tests auffinden.

Für jeden Test lässt sich einstellen, wie oft er durchlaufen werden soll und wie lange zwischen den einzelnen Durchgängen pausiert werden soll. Damit eignet sich Zanebug hervorragend für Stresstests und Systemtests innerhalb komplexer Umgebungen, bei denen auf andere Systemteile gegebenenfalls gewartet werden muss.

Mit diesen Eigenschaften ausgestattet, bietet Zanebug eine ideale Infrastruktur um die Leistungsfähigkeit einzelner Systemteile zu untersuchen. Der integrierte Performance-Monitor erlaubt es, eine Vielzahl an Leistungsmerkmalen zu überwachen. Neben den typischen Metriken zum Speicherverbrauch, CPU-Last oder Netzwerkauslastung, lassen sich z.B. detaillierte Statistiken zur CLR (Common Language Runtime), den wichtigsten Netzwerkprotokollen und wichtigen Betriebssystemdiensten anzeigen. Abbildung 5.2 zeigt eine kleine Auswahl von TCP-Metriken nach der Ausführung einiger Tests.

Wie der Autor, Sean McCormack, in einem Artikel über die Entwicklung dieses Testwerkzeugs schreibt (siehe [McC]), stellte das .NET-Konzept der Anwendungsdomänen (AppDomains) die größte Herausforderung bei der Realisierung von Zanebug dar. Anwendungsdomänen sind isolierte Umgebungen, in

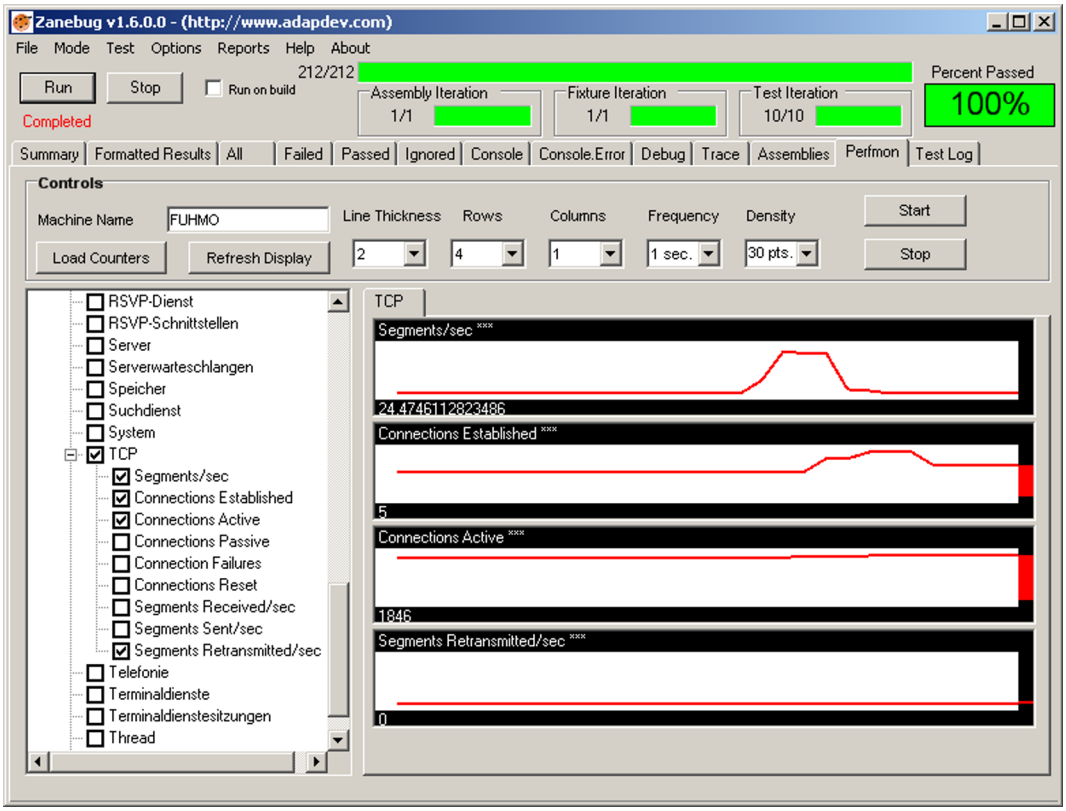


Abbildung 5.2: Performance-Monitor von Zanebug

denen Programme ausgeführt werden. Immer wenn ein Programm eine Assembly lädt, wird sie zur aktuellen Anwendungsdomäne hinzugefügt und vor Veränderungen geschützt. Damit trotzdem neue Versionen der geladenen dynamischen Bibliotheken erstellt werden können, ohne dass Zanebug dazu beendet werden muss, wird die zu testende Assembly kopiert und die Kopie geladen. Dabei galt es wohl einige Fallstricke zu überwinden und letztlich funktioniert dieser Mechanismus auch noch nicht fehlerfrei. Es kann passieren, dass Tests nach dem ersten Durchlauf nicht erneut gestartet werden können, erst nach einer Verzögerung starten oder dass das Programm nicht korrekt beendet wird. Das Programmfenster wird dann zwar geschlossen, der Prozess aber lebt weiter und muss manuell beendet werden.

5.2.3 CppUnit

CppUnit [SFL+] ist die erste Portierung von JUnit nach C++. Es ist quelloffen und lässt sich unter allen wichtigen Betriebssystemen einsetzen. CppUnit wird ausschließlich als Source-Code unter der LGPL-Lizenz [lgp] verbreitet.

Anders als bei den bisher vorgestellten Werkzeugen, kann bei CppUnit nicht mit Attributen gearbeitet werden. Die Runtime Type Information (RTTI) von C++ ermöglicht zwar einfache Typabfragen zur Laufzeit, weitergehende Reflection-Mechanismen wie sie zur Definition und Auswertung von Attributen nötig wären, fehlen jedoch. Deshalb verwendet CppUnit, wie viele andere Unit-Testing-Frameworks auch, das Konzept der Vererbung um dem Testrunner die Testklassen und Testfälle bekannt zu machen. In C++ würde der Testfall aus 5.2.1 wie in den Listings 5.2 und 5.3 aussehen.

Listing 5.2: MessageTest.h

```
#ifndef _MESSAGETEST_H_
#define _MESSAGETEST_H_
#include <cppunit/extensions/HelperMacros.h>
#include "AbstractStub.h"
#include "Message.h"
class MessageTest : public CPPUNIT_NS::TestFixture
{
        CPPUNIT_TEST_SUITE (MessageTest);
        CPPUNIT_TEST (SerializeTest);
        CPPUNIT_TEST_SUITE_END ();
private:
        Message* message;
public:
        void setUp();
        void tearDown();
protected:
        void SerializeTest();
};
#endif
```

Die Assertions sind in CppUnit als Makros definiert. `CPPUNIT_ASSERT_EQUAL(expected,actual)` kann zwei Werte vergleichen, wenn sie die folgenden Bedingungen erfüllen:

- Sie sind vom gleichen Typ.
- Sie können mit dem Operator << in einen `std::strstream` serialisiert werden.
- Sie können mit dem Operator == verglichen werden.

Daneben gibt es u.a. noch `CPPUNIT_ASSERT(condition)`, mit dem ein boolscher Wert überprüft

Listing 5.3: MessageTest.cpp

```
#include "MessageTest.h"
CPPUNIT_TEST_SUITE_REGISTRATION( MessageTest );
void MessageTest::setUp()
{
        this->message = new Message();
}
void MessageTest::tearDown()
{
        delete message;
}
void MessageTest::SerializeTest()
{
        this->message->WriteInt(43);
        this->message->SetSequenceNumber(7);
        byte expected[] = { 16, 0, 0, 0,  0, 0, 0, 0,  7, 0, 0, 0,  43, 0, 0, 0 };
        byte* result = this->message->Serialize();
        for(int i = 0; i < 16 ;i++)
        {
                CPPUNIT_ASSERT_EQUAL(expected[i], result[i]);
        }
}
```

wird und `CPPUNIT_FAIL(message)`, das immer fehl schlägt und die übergebene Nachricht ausgibt. Mit `CPPUNIT_ASSERT_THROW(expression, ExceptionType)` gibt es auch unter CppUnit die Möglichkeit zu testen, ob alle Ausnahmen wie erwartet ausgelöst werden. Die Testregistrierung findet in der Header-Datei mit dem Makro `CPPUNIT_TEST(testMethod)` statt.

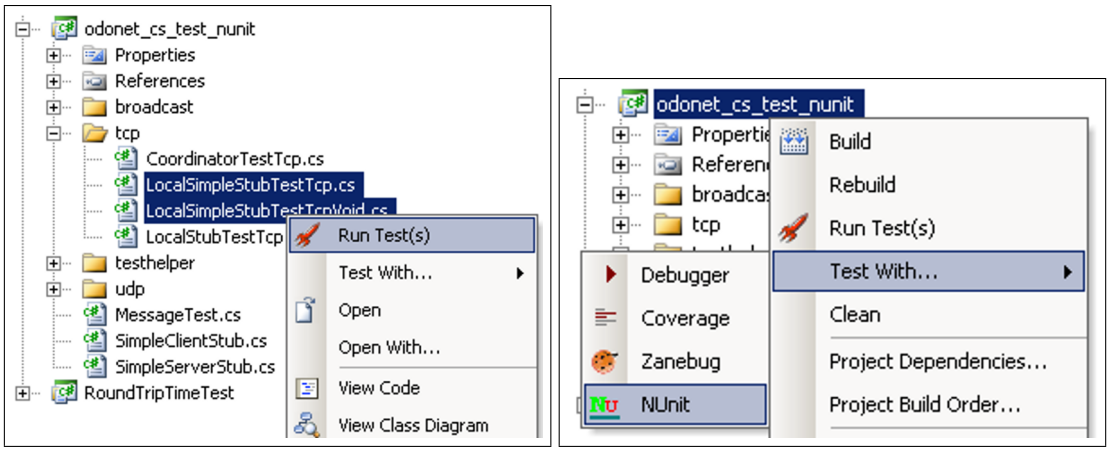
Um die Tests ausführen zu können, muss dem Projekt noch die TestRunner-Klasse hinzugefügt werden, die ohne Änderungen aus den mitgelieferten Beispielen verwendet werden kann. Damit lässt sich dann eine einfache Anwendung erstellen, die die Testergebnisse auf der Konsole ausgibt. Zur Erstellung einer Testanwendung mit graphischer Oberfläche gibt es für Windows die Klasse MfcTestRunner oder alternativ die vom Quasar Toolkit von Trolltech abhängige Klasse QtTestRunner. Beide Klassen ermöglichen es, Testfälle relativ einfach in einem ausführbaren Programm zusammenzufassen. Für das Testen dynamischer Bibliotheken wird ein Kommandozeilenprogramm mitgeliefert, das die Testergebnisse in eine XML-Datei schreiben kann.

Zusammenfassend lässt sich sagen, dass mit CppUnit, obwohl es lange nicht so komfortabel zu benutzen ist wie die vorgestellten Unit-Testing-Frameworks für C#, recht einfach umfangreiche Tests formuliert, durchgeführt und protokolliert werden können.

5.2.4 TestDriven.NET

TestDriven.NET [tesa] von Mutant Design Limited ist kein Unit-Testing-Framework, sondern ein Add-In für Visual Studio .NET, das viele andere Werkzeuge für Unit Tests in die Entwicklungsumgebung integriert. Eine Einzellizenz des ehemals NUnitAddIn genannten Werkzeugs kostet für den kommerziellen Einsatz 135 €. Für Studenten, Open Source Entwickler oder zum Ausprobieren ist die Software kostenlos verfügbar[3]. Aktuell ist die Version 2.0 RC1, die auch die Integration in Visual Studio 2005 voll unterstützt. Laut Herstellerangaben lassen sich alle wichtigen Unit-Testing-Frameworks integrieren, darunter auch NUnit, Zanebug und Visual Studio Team System. Tatsächlich werden die populärsten Werkzeuge bereits unterstützt und die kontinuierliche Weiterentwicklung wird die Zusammenarbeit mit weiteren Tools ermöglichen [Can].

TestDriven.NET ermöglicht es, Tests direkt über das Kontextmenü von Visual Studio auszuführen. Damit entfällt das Hin- und Herwechseln zwischen Entwicklungsumgebung und Testwerkzeug, was wiederum den Entwicklungsprozess beschleunigt. Einzelne Testmethoden können über das Kontextmenü des Texteditors aufgerufen werden, sollen mehrere Testklassen oder ganze Testprojekte durchlaufen werden, geschieht das über das Kontextmenü des Solution Explorers (Abbildung 5.3(a)).



(a) Run Test(s) (b) Test With...

Abbildung 5.3: Kontextmenü TestDriven.NET

Neben dem „RunTest(s)“-Eintrag gibt es noch den Menüpunkt „Test With...“ mit dem Tests mit dem Debugger durchlaufen werden können. Ist das Tool NCover [nco] installiert, kann man über dieses Menü die Testabdeckung (Code Coverage) überprüfen. Über das Kontextmenü von Testprojekten können außerdem die unterstützten externen Testwerkzeuge aufgerufen werden, die dann automatisch die neueste Version der zugehörigen Bibliothek übergeben bekommen (Abbildung 5.3(b)).

[3]Diese Information findet sich auf der Website aber gar nicht mehr so leicht [tesb].

5.2.5 Visual Studio Team System

Microsoft bietet mit dem Visual Studio Team System [vis] eine umfangreiche Sammlung von Werkzeugen für das Visual Studio 2005 an. Neben der sehr teuren Visual Studio Team Suite (ca. 12000 €), die die Entwicklungsumgebung und alle Werkzeuge umfasst, gibt es Pakete, die speziell auf die Bedürfnisse von Softwarearchitekten, -entwicklern oder -testern zugeschnitten sind und jeweils nur eine Teilmenge aller Features enthalten. Für den stolzen Preis von 6000 € bekommt der Software-Tester Werkzeuge, mit denen sich mit minimalem Aufwand Unit-Tests, Tests für Webanwendungen, Lasttests oder generische Tests für externe Anwendungen erstellen und durchführen lassen. Prinzipiell besteht die Möglichkeit, all diese Tests auch mit frei verfügbaren oder günstigeren Tools von Drittanbietern durchzuführen. Der wesentliche Vorteil der Microsoft-Lösung ist aber, dass alles aus einem Guss ist und deshalb eine höhere Produktivität erreicht werden kann. Konfigurations- oder Kompatibilitätsprobleme, wie sie bei den Werkzeugen verschiedener Hersteller auftreten können, werden so vermieden.

5.3 Testplanung

Die Testplanung soll klären, welche Tests benötigt werden. Das Ziel dabei ist aber nicht, im Voraus jeden einzelnen Testfall zu identifizieren, sondern vielmehr mögliche Teststrategien aufzuzeigen, die ein schnelles, risikoorientiertes Testen der Software ermöglichen.

Beim Unit-Testing (5.1.1) ergeben sich die Testfälle aus den Methoden der zu testenden Unit, die mit möglichst vielfältigen Parameterkombinationen aufgerufen werden sollten. Zuerst sollte immer der „Best Case“ getestet werden. In diesem Fall sind alle Übergabeparameter vorhanden und besitzen gültige Werte. Möglicherweise empfiehlt es sich, auch die minimal und maximal zulässigen Werte für bestimmte Parameter zu überprüfen („Minimum Case“ und „Maximum Case“). Im „Error Case“ wird die Reaktion der Funktion auf fehlende oder ungültige Werte getestet. Ob es ausreicht, die öffentliche Schnittstelle einer Klasse zu testen oder ob auch private Methoden überprüft werden sollen, ist unter Entwicklern umstritten. Das Testen privater Methoden bedeutet zunächst einen Mehraufwand, weil sie nicht direkt aufgerufen werden können. Bei C++ hält sich dieser aber in Grenzen: Die Sichtbarkeit der Methoden einer Klasse lässt sich leicht ändern, weil nicht vor jeder einzelnen Deklaration angegeben wird, ob die Methode öffentlich, geschützt oder privat sein soll. Statt dessen werden sie in der Header-Datei zu Sektionen, die die Sichtbarkeit bestimmen, zusammengefasst. Die Schnittstelle einer Klasse kann dann einfach verändert werden, indem mit dem Präprozessor geschützte oder private Sektionen öffentlich gemacht werden. Bei C# hingegen wäre eine bedingte Kompilierung aufwändiger, weil bei jeder Methode Code für den Präprozessor eingefügt werden müsste. Alternativ dazu könnte man das .NET Reflection API benutzen, um nicht öffentliche Methoden aufzurufen. Wird bei der Kompilierung einer C#-Assembly angegeben, dass sie Reflection unterstützen soll, wird sie mit Metadaten angereichert, die es u.a. ermöglichen, zur Laufzeit alle enthaltenen Methoden zu ermitteln und aufzurufen. Diese Vorgehensweise ist allerdings sehr unpraktisch wenn viel Refactoring betrieben wird, weil alle Reflectionmethoden Strings entgegennehmen, die den Bezeichnungen der Eigenschaften, Methoden oder Klassen entsprechen müssen. Werden diese Namen geändert, müssen auch alle übergebenen Strings angepasst werden. Visual Studio 2005 bietet beim Umbenennen zwar an, nach Zeichenketten zu suchen, die von dieser Änderung betroffen sein könnten, diese Suche ist aber nicht intelligent, so dass immer noch Handarbeit für den Programmierer bleibt. Ein weiterer Grund gegen das Testen privater Methoden ist, dass i.A. angenommen werden kann, dass wenn beim Testen der öffentlichen Methoden einer Klasse keine Fehler auftreten, wohl auch die privaten Methoden korrekt funktionieren müssen[4]. Der große Vorteil beim Testen privater Methoden ist allerdings, dass eventuell auftretende Fehler schneller lokalisiert werden können.

Isolierte Unit- und Integrationstests sind für die Netzwerkbibliothek allerdings schwierig zu realisieren.

[4] Denn was ist der Zweck einer Funktion, wenn sie niemals aufgerufen wird?

Durch die Schichtenarchitektur sind die meisten Klassen von anderen Klassen abhängig. Zwar könnten Mock-Objekte dazu verwendet werden, an den Schnittstellen zwischen den Schichten die jeweils darunterliegende Schicht zu ersetzen, mit Mock-Objekten kann jedoch nur überprüft werden, ob die richtigen Funktionen mit den richtigen Parametern aufgerufen wurden. Soll das Verhalten über Threadgrenzen hinweg untersucht werden, im Fall der Netzwerkbibliothek also z.B. nach dem Versenden einer Nachricht über einen Ersatz-NetworkClient auf das Eintreffen einer Antwort gewartet werden, kann nicht mehr mit Mocking-Frameworks gearbeitet werden. Statt dessen müssten manuell Ersatzimplementierungen (Stubs) geschaffen werden, die auf eine ebenfalls eigens für diesen Zweck implementierte Netzwerkemulation zugreifen. Damit ließe sich ohne großen Aufwand ein direkter Austausch von Nachrichtenreferenzen mittels einer statischen Klasse realisieren. Bei der Nachahmung komplexerer Aufgaben, etwa dem Auf- oder Abbau von Verbindungen oder der Emulation des Zeitverhaltens , muss allerdings hinterfragt werden, ob der dazu notwendige Aufwand noch in einem vernünftigen Verhältnis zum Nutzen steht.

Da die Netzwerkbibliothek in C# und in C++ implementiert und getestet werden muss, erscheint der zusätzliche Aufwand für das Testen in Isolation recht hoch. Zusätzlich zur Einarbeitungszeit in Mocking-Frameworks für C# und C++ kommt noch hinzu, dass Mono noch nicht voll kompatibel zu Microsofts .NET 2.0 ist. Der Einsatz einer Mocking-Lösung für C# unter Linux könnte dadurch erheblich erschwert werden. Insgesamt erscheint es also sinnvoller, nur solche Klassen isoliert zu testen, bei denen das ohne größere Änderungen am Programmcode und ohne Mock-Objekte oder Netzwerkemulation möglich ist. Die Fehlerfreiheit soll also im Wesentlichen mit Systemtests sichergestellt werden, was zwar einerseits die Fehlerlokalisierung erschwert, auf der anderen Seite den Aufwand für das Testen der Netzwerkbibliothek aber erheblich reduziert. Die einzelnen Testfälle können aus der Anforderungsanalyse abgeleitet werden. Es ist also zu prüfen, ob die Verbindungen wie gewünscht aufgebaut werden, ob alle unterstützten Datentypen (3.1.1) fehlerfrei von den verschiedenen Transporttypen (3.2.1) übertragen werden und ob Nachrichten auf entsprechende Funktionsaufrufe abgebildet werden, welche im Falle einer Zeitüberschreitung den festgelegten Fehlerwert zurückliefern (3.1.1). Dabei muss auch das Verhalten bei größeren Nachrichten untersucht werden. Außerdem ist in jedem Fall noch zu prüfen, ob die Kommunikation zwischen den Instanzen und dem Kontrollzentrum reibungslos funktioniert (2.2.3).

5.4 Testdurchführung

Wie aus dem letzten Kapitel hervorgeht sind die meisten Tests nicht als Unit-Tests, sondern als Systemtests geplant. Trotzdem können die meisten Testfälle mit den in 5.2 vorgestellten Werkzeugen recht einfach realisiert werden. Für die Tests in C# ist jedoch zu beachten, dass Zanebug gar nicht unter Mo-

no lauffähig ist und NUnit nur in der Version 2.2.0 weit verbreitet ist. Der aktuellen Version 2.4 Beta 1 wurde jedoch ein NAnt-Build-Script hinzugefügt, mit dem sich ein NUnit-Konsolen-Testrunner für Mono 2.0 erstellen lässt. Das Fehlen eines graphischen Frontends lässt sich verschmerzen, wenn dafür das Programmieren der Tests durch die neuen Assertions und Attribute erleichtert wird. Um die Tests unter Windows sowohl mit NUnit, als auch mit Zanebug, das vor allem für Last- oder Performanztests interessant ist, durchführen zu können, müssen die Testprojekte auf die NUnit-Version 2.2.7 beschränkt werden, mit der Zanebug voll kompatibel ist. Durch die Abwärtskompatibilität der neueren Versionen können die Testprojekte dann auch ohne Änderungen unter Mono verwendet werden. Für das Testen der C++-Version wird ebenfalls nur die einfache Konsolenlösung von cppUnit verwendet. Alternativ wäre noch eine Implementierung mit einer graphischen Benutzeroberfläche, die auf der für Windows und Linux verfügbaren Qt-Bibliothek aufsetzen müsste, in Frage gekommen. Eine einfache Qt-Testrunner-Implementierung bringt jedoch wenig Vorteile, so dass der Aufwand für die Installation und die Einarbeitung in das Qt-Toolkit zu hoch erscheint.

Für das Unit-Testing eignen sich besonders die Klassen `Message` und `NetworkConfigEntry`. Die Klasse `NetworkConfigEntry` hat die statische Methode `Validate(INetworkConfigEntry)`, die überprüfen soll, ob ein Konfigurationsobjekt alle benötigten Informationen enthält und fehlende Werte so weit wie möglich mit voreingestellten Werten belegt oder Ausnahmen auslöst falls dies nicht möglich ist. Die Unit-Tests dieser Klasse bestehen im Wesentlichen darin, zu überprüfen, ob ungültige oder fehlende Werte als solche erkannt werden und ob die richtigen Default-Werte eingesetzt werden, bzw. ob alle erwarteten Ausnahmen mit aussagekräftigen Fehlermeldungen ausgelöst werden. Bei der `Message`-Klasse wurden neben den Konstruktoren alle Methoden zum Lesen oder Schreiben der unterstützten Datentypen und die Methoden zum Serialisieren und Deserialisieren getestet (Listing A.3). Ungetestet blieben lediglich einfache Properties bzw. Getter- und Setter-Methoden. Damit wurden 94% der Methoden dieser Klasse getestet. Abbildung 5.4 zeigt das Fenster des zur Feststellung der Testabdeckung verwendeten Werkzeugs NCover [nco].

Beim Entwurf der Systemtests ist zu beachten, dass die Kommunikation zwischen den Implementierungen in C# und C++ getestet werden können muss. Dazu wurden verschiedene Testprojekte für die verschiedenen Programmiersprachen angelegt, je eines für lokale Tests, und je ein Client- bzw. Serverprojekt für entfernte Tests. In den lokalen Testprojekten finden sich neben den Unit-Tests Systemtests, bei denen ein lokaler Server und ein lokaler Client erzeugt werden, die dann über das Loopback-Device miteinander kommunizieren. Bei den Testprojekten für entfernte Tests wird entweder ein Server oder ein Client erzeugt. Die Serverprojekte beinhalten allerdings gar keine Testfälle. Es wird lediglich ein Server erzeugt, der eingehende Anfragen bearbeitet. Die eigentlichen Tests finden in den Clientprojekten statt. Dort wird zunächst ein Client mit einem vorher gestarteten Server verbunden. Anschließend werden vom

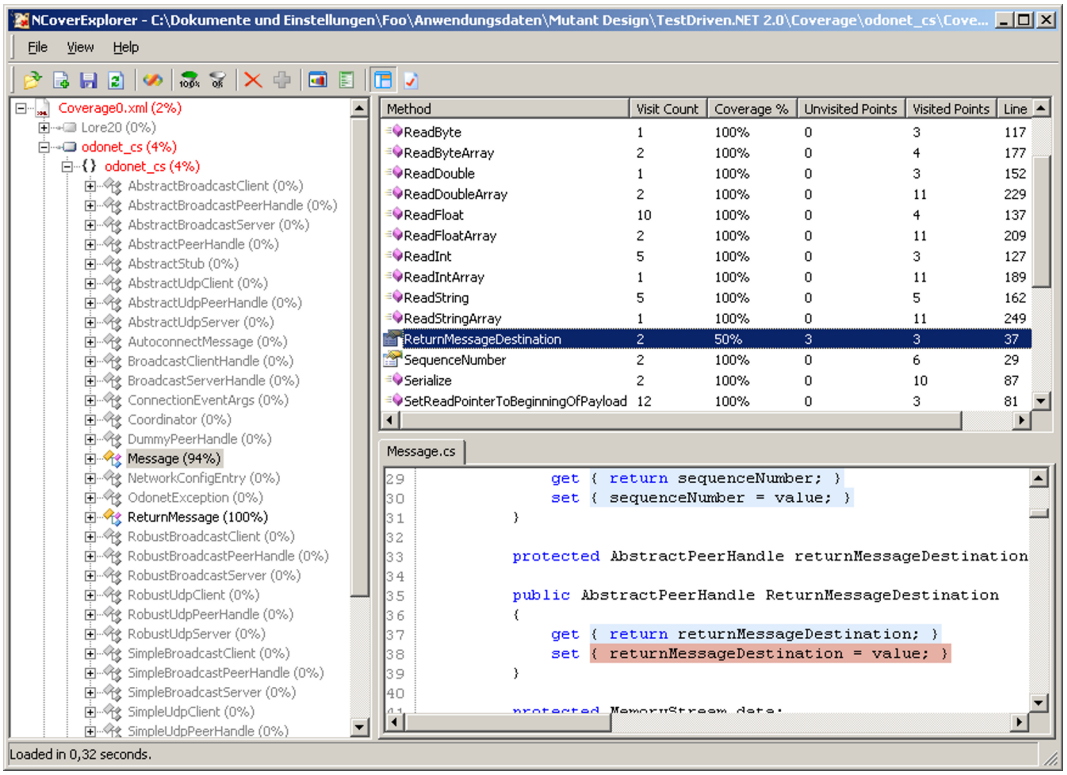


Abbildung 5.4: Testabdeckung der Klasse `Message` mit NCover

Client verschiedene Anfragen an den Server gestellt, deren Ergebnis dann mit dem erwarteten Ergebnis verglichen wird.

Da der Benutzer der Netzwerkbibliothek die Möglichkeit hat, sie um spezialisierte Schnittstellen zu erweitern, muss die für die Tests verwendete Schnittstelle möglichst allgemein gehalten werden. Dazu wurde das Interface `Simple` definiert, das Methoden zum Setzen und zum Auslesen aller unterstützten Datentypen und Rückgabewerte anbietet (Listing A.2). Der daraus erzeugte ServerStub sendet in den Basistests einzelne, zufällige Werte der verschiedenen Datentypen an den ClientStub, welcher sie in einer Hashtabelle bzw. einer `std::map` speichert. Anschließend wird der zuvor übertragene Wert vom Client wieder abgefragt und überprüft, ob beide Werte gleich sind. Die Speicherung in einer Hashtabelle ist nötig, weil der Server einen Fehlerwert zurückliefern muss, falls nach einem Wert gefragt wird der vorher gar nicht übertragen wurde. Dies könnte z.B. der Fall sein, wenn zur Übertragung des Wertes vom Client zum Server eine Funktion mit Rückgabetyp `void` verwendet wird und die dazugehörige Nachricht verloren ging oder ihre Interpretation und Ausführung auf Serverseite noch nicht abgeschlossen ist[5]. Um zwischen fehlenden Werten und Zeitüberschreitungen unterscheiden zu können, werden verschiedene Fehlerwerte verwendet. Daneben gibt es Tests, die große Nachrichten versenden um das Verhalten

[5]Wenn die Anfragen auf Serverseite nicht in der Reihenfolge abgearbeitet werden in der sie ankommen, muss das natürlich als Bug angesehen werden. Während der Implementierung der Netzwerkschicht ist aber genau dieser Fehler aufgetreten, weshalb für diesen Fall die Tests erweitert werden mussten.

der NetworkPeers zu untersuchen, wenn die Nachrichtengröße die Puffergröße der Sockets übersteigt. Um zu testen, ob bei einer Zeitüberschreitung auch wirklich der festgelegte Fehlerwert zurückgegeben wird, sendet der Client eine Anfrage an den Server, welcher in diesem Fall aber erst nach Ablauf des Timeout-Intervalls antwortet, so dass beim Client der Fehlerwert erwartet werden kann. Neben diesen Basistests wurde ebenfalls geprüft, ob auch bei einer großen Zahl von Clients der Abbau der Verbindungen und das Freigeben der Ressourcen problemlos von statten geht. Mit Zanebug wurde das Verhalten der NetworkPeers unter Stress getestet indem die einzelnen Tests sehr oft wiederholt wurden.

Bei der Implementierung der Testfälle wurden zunächst für jeden Transporttyp eigene Testklassen verwendet, um ohne Konfigurationsdateien arbeiten zu können. Daraus resultierte jedoch eine starke Codeduplikation und damit letzlich ein erhöhter Wartungsaufwand und eine zusätzliche Fehlerquelle. Außerdem ist es mit den einfachen Konsolen-Testrunnern schwierig einzelne Tests auszuwählen, so dass sich die Konfiguration über Dateien als sinnvoller erwiesen hat. Damit wird nur eine Implementierung der Testfälle benötigt. Welcher Transporttyp getestet werden soll, wird vor der Testausführung über Konfigurationsdateien festgelegt. Um die Implementierung der einzelnen Testklassen möglichst einfach zu halten und auch dort Codeduplikation zu vermeiden, wurden die Klassen `Testhelper` und `SimpleStubTestHelper` eingeführt. Die `Klasse` Testhelper vereinfacht u.a. den Zugriff auf Reflectionmethoden, wohingegen ein `SimpleStubTestHelper` im Wesentlichen die Implementierungen der Callback-Methoden des `SimpleClientStub` enthält.

Die Implementierungen der Schnittstelle des Kontrollzentrums wurde mit einfachen Anwendungen getestet. Im Hinblick auf den komplizierteren Entwurf mit Client- und ServerStub für jeden Kommunikationspartner, bei dem die Verbindungsverwaltung den Hauptunterschied zu den anderen Kommunikationsendpunkten ausmacht, ist eine eigenständige Anwendung die weniger aufwändige Lösung. Außerdem stellen diese Testanwendungen prototypische Implementierungen für das Kontrollzentrum dar, die dann um den zusätzlich benötigten Funktionsumfang erweitert werden können.

5.5 Testergebnisse

Durch das automatisierte Testen der Netzwerkbibliothek konnten Fehler gefunden und eleminiert werden, die von manuellen Tests vermutlich gar nicht erfasst worden wären. Vor allem Fehler, die durch Nebenläufigkeiten im Programmfluss entstehen, sind durch herkömmliche Testmethoden kaum reproduzierbar. Durch wiederholte Stresstests konnten Synchronisationsfehler und Probleme beim Freigeben von Ressourcen provoziert werden, die unter Normalbedingungen nur äußerst selten entstehen würden, deren Auswirkungen aber bis zum Absturz der Anwendung reichen können.

Die Lokalisierung dieser Fehler war aber trotz zahlreicher, aussagekräftiger Debug-Ausgaben im Programmcode der Netzwerkbibliothek oft sehr schwierig. Um das Zusammenspiel des Hauptthreads mit den Empfangsthreads beim Debuggen einigermaßen kontrollieren zu können, sind mehrere Debugger-Instanzen unverzichtbar, am besten sogar auf unterschiedlichen Rechnern. Erschwerend kamen beim Debuggen der Netzwerkbibliothek noch die Zeitüberschreitungen hinzu, so dass die Konfiguration je nach Situation angepasst werden musste. In jedem Fall erfordert die Suche nach Multithreading-bedingten Fehlern eine genaue Kenntnis des Programmcodes, damit einzelne Debug-Schritte immer in einer sinnvollen Reihenfolge ausgeführt werden können.

Durch die Verwendung eines Mocking-Frameworks zum isolierten Testen einzelner Units hätten sich viele Fehler bestimmt leichter lokalisieren lassen. Die unterschiedlichen Werkzeuge für C++ und C# und die unvollständige Implementierung von Mono 2.0 stellten das Verhältnis von Aufwand und Nutzen im Fall der Netzwerkbibliothek jedoch in Frage. Soll eine Software aber auf eine Programmiersprache und eine Plattform beschränkt bleiben, ließen sich durch diesen Ansatz, auch wenn er zunächst recht aufwändig erscheinen mag, vermutlich viele Debug-Sessions sparen. Übertragen auf das isolierte Testen einzelner Module des verteilten Simulations- und VR-Systems könnten Mock-Objekte an Stelle der ServerStubs verwendet werden, um den gesamten Dienst einer Komponente nachzuahmen. Ein neues Modul ließe sich so unabhängig von anderen, noch nicht implementierten oder potentiell fehlerhaften Modulen, entwickeln und testen.

6 Ergebnisse

Die im Rahmen dieser Arbeit entstandene Middleware ermöglicht es, Simulations- und VR-Anwendungen ohne großen Aufwand auf verschiedene Rechner zu verteilen. Dazu können, ähnlich wie beim Remote Procedure Call, anwendungsspezifische Kommunikationsendpunkte generiert werden, die es ermöglichen, entfernte Funktionsaufrufe weitgehend wie lokale Funktionsaufrufe aussehen zu lassen. Der Erstellungsprozess für diese Kommunikationsendpunkte wurde größtenteils automatisiert, so dass sich aus der Schnittstellenbeschreibung in XML direkt die entsprechenden dynamischen Bibliotheken erzeugen lassen, welche dann unmittelbar verwendet werden können.

Die ursprüngliche Netzwerkbibliothek für C++ von Joscha Metze wurde komplett überarbeitet und um eine Implementierung in C# erweitert. Damit wurden zusätzlich zu Windows und Linux .NET 2.0 und Mono 2.0 als Plattformen für die Entwicklung neuer Komponenten erschlossen. Neben der Abstraktion von der Programmiersprache wurde durch den neuen Entwurf auch die Integration neuer Netzwerktechnologien wie z.B. ATM oder Myrinet ermöglicht. Weiterhin wurden spezielle Kommunikationsendpunkte implementiert, mit denen über das Kontrollzentrum alle Module einer verteilten Anwendung überwacht und gesteuert werden können.

Bei der Schnittstellendefinition können nun funktionsspezifische Timeout-Intervalle und Fehlerwerte angegeben werden, so dass erstmals eine angemessene Behandlung von Zeitüberschreitungen ermöglicht wird. Daneben gestatten zahlreiche neue Konfigurationsparameter, mit denen z.B. die Sende- und Empfangspuffergrößen oder die zu verwendende Netzwerkkarte festgelegt werden können, eine bessere Anpassung an die Systemumgebung.

Die wichtigste Neuerung jedoch ist die ausführliche Dokumentation. Neben aussagekräftigen Kommentaren im Quellcode sind begründete Entwurfsentscheidungen und Erklärungen zu wichtigen Aspekten der Implementierung unverzichtbar für die Weiterentwicklung der Netzwerkbibliothek durch andere Personen. Ohne diese Hilfsmittel ist der Einarbeitungsaufwand immens, weil auf eigentlich längst beantwortete Fragen neue Antworten gefunden werden müssen.

7 Ausblick

Viele Unzulänglichkeiten der alten Netzwerkkommunikationsschicht wurden bei der Neuimplementierung beseitigt. Dennoch konnten aus Zeitgründen nicht alle Ideen zur Verbesserung umgesetzt werden.

Da die Bibliothek nach wie vor ausschließlich für den Einsatz in lokalen Netzen ausgelegt ist, die Berücksichtigung aller Sicherheitsfragen, die die Kommunikation über das Internet betreffen, aber nicht sinnvoll erscheint[1], sollte die Kommunikation über ein Wide Area Network über ein virtuelles privates Netzwerk (VPN) statt finden. Welches Protokoll (PPTP, IPSec, ...) sich dabei am besten eignet ist noch zu klären[2].

Neben kleineren Optimierungen und der Integration weiterer Netzwerktechnologien wären auch komplexere Implementierungen der NetworkPeers für den Zugriff auf die Transportschicht denkbar. So wäre z.B. eine zuverlässige Broadcast-Implementierung für die differentielle Datenübertragung über einen Broadcast-Kanal wünschenswert. Dazu müssten auf Empfängerseite alle Pakete aller unvollständigen Nachrichten zwischengespeichert und fehlende Pakete über einen separaten Kommunikationskanal erneut angefordert werden. Der Sender könnte die Paketwiederholungen über dieselbe zusätzliche Verbindung zustellen oder aber über den Broadcast-Kanal an alle verteilen.

Für die Entwicklung neuer Module wurde begleitend zu dieser Arbeit ein einfaches Simulationsframework geschaffen, mit dem ohne großen Aufwand neue Anwendungen erstellt werden können. Mit dem DirectInputMapper von Robert Bürger als Eingabekomponente, der Visualisierung von Joscha Metze als Ausgabekomponente und einer generischen Simulationsanwendung dazwischen, kann sich der Entwickler ganz auf die Programmierung seiner Simulation beschränken. Neue Simulationen implementieren dazu das Interface `ISimulationPlugin`, das die zwei Methoden `InputChanged` und `DoStep` definiert. Beide Methoden werden vom Framework aufgerufen, wobei `InputChanged` die neuen Eingaben übergeben bekommt und in `DoStep` genau ein diskreter Simulationsschritt implementiert werden soll. Unmittelbar nach jedem Aufruf von `DoStep` wird das Ergebnis an die Visualisierung übertragen. Als DLL erstellt, können die Simulationsplugins dann von der Simulationsanwendung (GenericSimu-

[1] Die Benutzung würde erschwert, die Performanz würde verringert und die Implementierung wäre aufwändig und fehleranfällig.

[2] Beim Point-to-Point-Tunneling-Protocol kann es je nach Konfiguration Probleme mit dem Broadcasting geben, IPSec hingegen ist in der Konfiguration recht aufwändig.

lation.exe) geladen und ausgeführt werden. Eine erste Verwendung fand dieses Framework bei einer Praktikumsarbeit zur Vorlesung „VR und Simulation“ von Prof. Dr. Markus Wacker an der HTW Dresden. Abbildung 7.1 zeigt das Ergebnis von René Schulte, René Janovsky und Torsten Bär, ein virtuelles Dosenschießen mit schneller Physiksimulation.

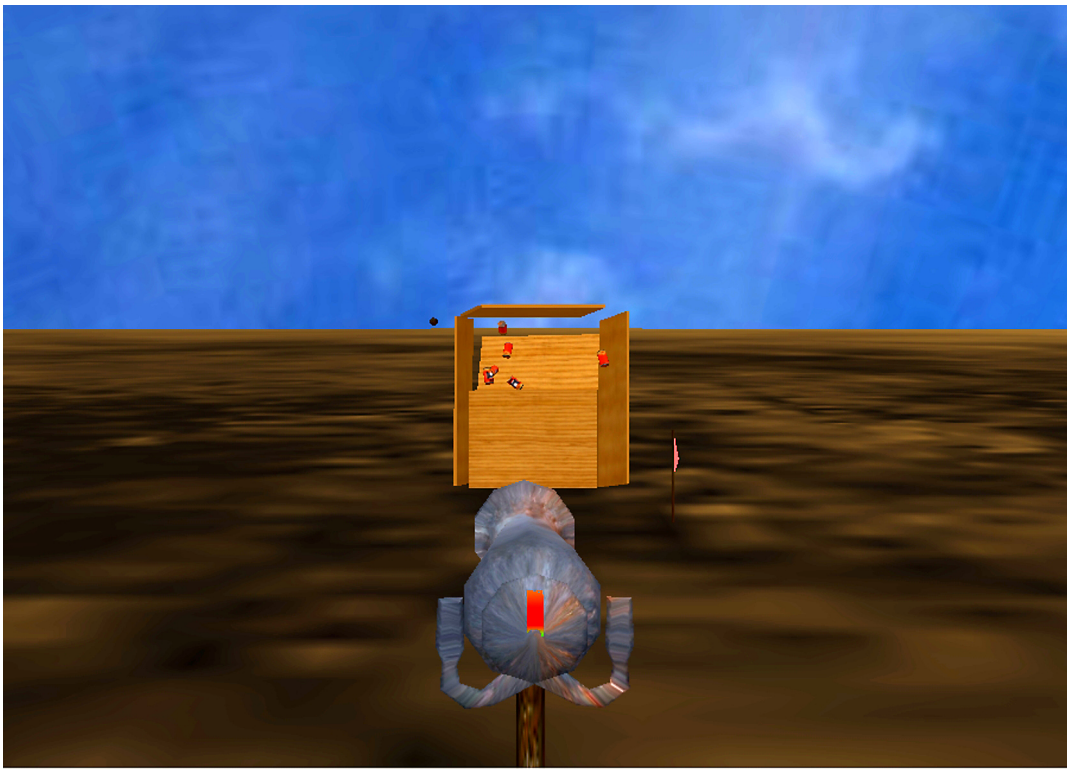


Abbildung 7.1: Virtuelles Dosenschießen

Gelingt es trotz der Verteilung auf verschiedene Computer nicht, bestimmte Simulationen in Echtzeit durchzuführen, könnte eine Anwendung entwickelt werden, die die Ausgabe der Simulationskomponente aufzeichnen und danach mit veränderter Geschwindigkeit abspielen kann. Wenn diese Anwendung dann den gleichen ServerStub wie die Simulationskomponente und den gleichen ClientStub wie die Visualisierungskomponente verwendet, kann sie zwischen diesen beiden Modulen zugleich als Relay agieren, so dass auch während der Aufzeichnung der Simulationsergebnisse eine Visualisierung stattfinden kann. Die Callback-Funktionen des ClientStubs müssten dann so implementiert werden, dass ihr Aufruf mit Zeitstempel und allen übergebenen Parametern protokolliert wird und dass wiederum die entsprechende ServerStub-Funktion aufgerufen wird. Beim Abspielen würde diese Anwendung dann die Simulationskomponente ersetzen und an Hand der gespeicherten Zeitstempel und Parameter direkt mit der Visualisierungskomponente kommunizieren.

Noch wird die neue Netzwerkbibliothek noch nicht von allen bestehenden Modulen verwendet, welche Erweiterungen letztlich wirklich benötigt werden wird erst der Praxiseinsatz zeigen.

Literaturverzeichnis

[apa] OPEN SOURCE INITIATIVE (Hrsg.). *Apache License, Version 2.0*.
http://www.opensource.org/licenses/apache2.0.php

[Beu] BEUST, Cédric. *Why unit tests are disappearing*.
http://beust.com/weblog/archives/000319.html

[blo] CATALYST DEVELOPMENT (Hrsg.). *Introduction to TCP/IP - Blocking vs. Non-Blocking Sockets*.
http://www.developerfusion.co.uk/show/28/8/

[bsd] OPEN SOURCE INITIATIVE (Hrsg.). *The BSD License*.
http://www.opensource.org/licenses/bsd-license.php

[Can] CANSDALE, Jamie. *Zero Friction Unit Testing for Visual Studio .NET*.
http://weblogs.asp.net/nunitaddin/

[CDK01] COULOURIS, George ; DOLLIMORE, Jean ; KINDBERG, Tim: *Distributed Systems - Concepts and Design*. 3. Addison-Wesley, 2001 (ISBN 0-201-61918-0)

[cor] OBJECT MANAGEMENT GROUP (Hrsg.). *CORBA*.
http://www.corba.org/

[Fea] FEATHERS, Michael. *A Set of Unit Testing Rules*.
http://www.artima.com/weblogs/viewpost.jsp?thread=126923

[Fow] FOWLER, Martin. *Mocks aren't Stubs*.
http://www.martinfowler.com/articles/mocksArentStubs.html

[haw] HAWK SOFTWARE (Hrsg.). *HawkNL™(Hawk Network Library)*.
http://www.hawksoft.com/hawknl/

[How] HOWER, Chad Z. *Introduction to Indy*.
http://www.swissdelphicenter.ch/en/showarticle.php?id=4

[iee98] *IEEE Standard for Distributed Interactive Simulation - Application Protocols*. The Institute of Electrical and Electronics Engineers, Inc., 1998 (ISBN 0-7381-0174-5)

[Kap03] KAPPLER, Thomas: *Quality of Service und Internet-Telefonie*. http://www.stud.uni-karlsruhe.de/~uphr/files/Quality%20of%20Service%20und%20Internet-Telefonie/ausarbeitung.html: Universität Karlsruhe, February 2003

[KBP01] KANER, Cem ; BACH, James ; PETTICHORD, Bred: *Lessons learned in Software Testing*. 1. John Wiley & Sons, 2001. – ISBN 0471081124

[lgp] OPEN SOURCE INITIATIVE (Hrsg.). *GNU Lesser General Public License*. http://www.opensource.org/licenses/lgpl-license.php

[lib] SCHULZE, Martin (Hrsg.) ; CUMMING, Murray (Hrsg.). *libsigc++ – The Typesafe Callback Framework for C++*. http://libsigc.sourceforge.net/

[McC] MCCORMACK, Sean. *Zanebug – The development of an automated unit testing tool*. http://linux.sys-con.com/read/105673_1.htm

[Met05] METZE, Joscha. *Entwicklung eines Visualisierungssystems für Virtual-Reality Simulationen*. 2005

[MSD06] MSDN: *Echtzeitverhalten von .NET Compact Framework*. http://www.microsoft.com/germany/msdn/library/net/compactframework/EchtzeitverhaltenVonNETCompactFramework.mspx?mfr=true: Microsoft, April 2006

[nco] CODE MORTEM, INC (Hrsg.). *NCover - Code Coverage for .NET*. http://ncover.org/

[New] NEWMAN, Sam. *Why unit tests are important*. http://www.magpiebrain.com/blog/2005/09/18/why-unit-tests-are-important/

[NMW05] NEIDHOLD, Benjamin ; METZE, Joscha ; WACKER, Markus: Towards a general concept for distributed visualisation of simulations in Virtual Reality environments. In: *Proceedings of Virtual Environments* Eurographics/ACM, 2005

[Pac02] Packeteer, Inc.: *Packeteer Technical White Paper Series - Response-Time Technology*. May 2002

[Rä04] RÄTZMANN, Manfred: *Software-Testing & Internationalisierung*. 2. Galileo Computing, 2004. – ISBN 3–89842–539–8

[RFCa] IETF (Hrsg.). *An Architecture for Differentiated Services*. http://www.ietf.org/rfc/rfc2475.txt

[rfcb] IETF (Hrsg.). *Binding Protocols for ONC RPC Version 2*. http://www.ietf.org/rfc/rfc1833.txt

[RFCc] IETF (Hrsg.). *Integrated Services in the Internet Architecture: an Overview*. http://www.ietf.org/rfc/rfc1633.txt

[rfcd] IETF (Hrsg.). *RPC: Remote Procedure Call Protocol Specification Version 2 (Memo)*. http://www.ietf.org/rfc/rfc1057.txt

[rfce] IETF (Hrsg.). *RPC: Remote Procedure Call Protocol Specification Version 2 (Standards Track)*. http://www.ietf.org/rfc/rfc1831.txt

[rfcf] IETF (Hrsg.). *TCP Congestion Control*. http://www.ietf.org/rfc/rfc2581.txt

[rmi] SUN MICROSYSTEMS (Hrsg.). *Java Remote Method Invocation*. http://java.sun.com/products/jdk/rmi/

[RS] RÖHRL, Martin ; SCHMIEDL, Stefan. *Ich habe Tests, also kann ich!* http://www.dotnet-magazin.de/itr/online_artikel/psecom,id,252,nodeid,31.html

[sdl] SDL PROJECT (Hrsg.). *Simple DirectMedia Layer*. http://www.libsdl.org/index.php

[SFL+] SOMMERLADE, Eric ; FEATHERS, Michael ; LACOSTE, Jerome ; LEPILLEUR, Baptiste ; BAKKER, Bastiaan ; ROBBINS, Steve. *CppUnit Wiki*. http://cppunit.sourceforge.net/cppunit-wiki/FrontPage

[Sub] SUBBIAH, Palani. *Die Bitfehlerrate hinterfragt*. http://www.elektroniknet.de/index.php?id=1123

[Tan96] TANENBAUM, Andrew S.: *Computer networks*. 3. Prentice Hall Europe, 1996. – ISBN 0–13–394248–1

[tesa] MUTANT DESIGN LIMITED (Hrsg.). *TestDriven.NET*. http://www.testdriven.net/

[tesb] MUTANT DESIGN LIMITED (Hrsg.). *TestDriven.NET - Software License Terms*. http://www.testdriven.net/purchase_licenses.aspx

[TPCF] TWO, Michael C. ; POOLE, Charlie ; CANSDALE, Jamie ; FELDMAN, Gary. *NUnit*. http://www.nunit.org/

[vis] MICROSOFT (Hrsg.). *Visual Studio Team System*. http://msdn.microsoft.com/vstudio/teamsystem/default.aspx

[zan] ADAPDEV TECHNOLOGIES (Hrsg.). *Adapdev Technologies, LLC - Zanebug*. http://www.adapdev.com/zanebug/

[zli] OPEN SOURCE INITIATIVE (Hrsg.). *The zlib/libpng License*. http://www.opensource.org/licenses/zlib-license.html

A Listings

Listing A.1: Beispiel einer Netzwerkkonfiguration

```
<?xml version="1.0" encoding="iso-8859-1"?>
<configuration name="SimpleStubTestConfiguration"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:noNamespaceSchemaLocation="odoconfig.xsd">

  <section id="network">
    <object id="SimpleClientStubTcp" type="ClientStub">
      <attr name="type">TcpServer</attr>
      <attr name="port">3000</attr>
      <attr name="timeout">500</attr>
      <attr name="autoconnect">0</attr>
      <attr name="autoconnectport">4000</attr>
    </object>
    <object id="SimpleServerStubTcp" type="ServerStub">
      <attr name="type">TcpClient</attr>
      <attr name="servernames">localhost</attr>
      <attr name="port">3000</attr>
      <attr name="autoconnect">0</attr>
      <attr name="autoconnectport">4000</attr>
    </object>

    <object id="SimpleClientStubRobustUdp" type="ClientStub">
      <attr name="name" />
      <attr name="type">RobustUdpServer</attr>
      <attr name="port">3000</attr>
      <attr name="timeout">500</attr>
      <attr name="autoconnect">0</attr>
      <attr name="autoconnectport">4000</attr>
    </object>
    <object id="SimpleServerStubRobustUdp" type="ServerStub">
      <attr name="name" />
      <attr name="type">RobustUdpClient</attr>
      <attr name="servernames">localhost</attr>
      <attr name="port">3000</attr>
      <attr name="timeout">500</attr>
      <attr name="autoconnect">0</attr>
      <attr name="autoconnectport">4000</attr>
    </object>
```

```
<object id="SimpleClientStubSimpleUdp" type="ClientStub">
  <attr name="name" />
  <attr name="type">SimpleUdpServer</attr>
  <attr name="port">3000</attr>
  <attr name="timeout">500</attr>
  <attr name="autoconnect">0</attr>
  <attr name="autoconnectport">4000</attr>
</object>
<object id="SimpleServerStubSimpleUdp" type="ServerStub">
  <attr name="name" />
  <attr name="type">SimpleUdpClient</attr>
  <attr name="servernames">localhost</attr>
  <attr name="port">3000</attr>
  <attr name="timeout">500</attr>
  <attr name="autoconnect">0</attr>
  <attr name="autoconnectport">4000</attr>
</object>

<object id="SimpleClientStubRobustBroadcast" type="ClientStub">
  <attr name="name" />
  <attr name="type">RobustBroadcastServer</attr>
  <attr name="servernames" />
  <attr name="localname">localhost</attr>
  <attr name="port">3000</attr>
  <attr name="timeout">1000</attr>
  <attr name="autoconnect">0</attr>
  <attr name="nodelay">0</attr>
  <attr name="sendbuffersize">8192</attr>
  <attr name="receivebuffersize">8192</attr>
</object>
<object id="SimpleServerStubRobustBroadcast" type="ServerStub">
  <attr name="name" />
  <attr name="type">RobustBroadcastClient</attr>
  <attr name="servernames">localhost</attr>
  <attr name="localname">localhost</attr>
  <attr name="port">3000</attr>
  <attr name="timeout">1000</attr>
  <attr name="autoconnect">0</attr>
  <attr name="nodelay">0</attr>
  <attr name="sendbuffersize">8192</attr>
  <attr name="receivebuffersize">8192</attr>
</object>

<object id="SimpleClientStubSimpleBroadcast" type="ClientStub">
  <attr name="name" />
  <attr name="type">SimpleBroadcastServer</attr>
  <attr name="servernames" />
  <attr name="localname">localhost</attr>
```

```
      <attr name="port">3000</attr>
      <attr name="timeout">1000</attr>
      <attr name="autoconnect">0</attr>
      <attr name="nodelay">0</attr>
      <attr name="sendbuffersize">8192</attr>
      <attr name="receivebuffersize">8192</attr>
    </object>
    <object id="SimpleServerStubSimpleBroadcast" type="ServerStub">
      <attr name="name" />
      <attr name="type">SimpleBroadcastClient</attr>
      <attr name="servernames">localhost</attr>
      <attr name="localname">localhost</attr>
      <attr name="port">3000</attr>
      <attr name="timeout">1000</attr>
      <attr name="autoconnect">0</attr>
      <attr name="nodelay">0</attr>
      <attr name="sendbuffersize">8192</attr>
      <attr name="receivebuffersize">8192</attr>
    </object>
  </section>
</configuration>
```

Listing A.2: Schnittstellenbeschreibung die alle unterstützten Parameter- und Rückgabetypen beinhaltet

```
<?xml version="1.0" encoding="utf-8"?>
<messagetable name="Simple">
  <message name ="GetByte" id="1" error="255">
    <returntype type="byte" />
    <parameter type="int" name="valueID" />
  </message>
  <message name ="GetInt" id="2" error="-9999">
    <returntype type="int" />
    <parameter type="int" name="valueID" />
  </message>
  <message name ="GetFloat" id="3" error="-9999">
    <returntype type="float" />
    <parameter type="int" name="valueID" />
  </message>
  <message name ="GetDouble" id="4" error="-9999">
    <returntype type="double" />
    <parameter type="int" name="valueID" />
  </message>
  <message name ="GetString" id="5" error="error">
    <returntype type="string" />
    <parameter type="int" name="valueID" />
  </message>

  <message name ="GetByteArray" id="6" error="0|0|0">
    <returntype type="byte[]" />
```

```
  <parameter type="int" name="valueID" />
</message>
<message name ="GetIntArray" id="7" error="-9999|-9999|-9999">
  <returntype type="int[]" />
  <parameter type="int" name="valueID" />
</message>
<message name ="GetFloatArray" id="8" error="-9999|-9999|-9999">
  <returntype type="float[]" />
  <parameter type="int" name="valueID" />
</message>
<message name ="GetDoubleArray" id="9" error="-9999|-9999|-9999">
  <returntype type="double[]" />
  <parameter type="int" name="valueID" />
</message>
<message name ="GetStringArray" id="10" error="error">
  <returntype type="string[]" />
  <parameter type="int" name="valueID" />
</message>

<message name ="SetByte" id="11" error="0">
  <returntype type="int" />
  <parameter type="int" name="valueID" />
  <parameter type="byte" name="value" />
</message>
<message name ="SetInt" id="12" error="-9999">
  <returntype type="int" />
  <parameter type="int" name="valueID" />
  <parameter type="int" name="value" />
</message>
<message name ="SetFloat" id="13" error="-9999">
  <returntype type="int" />
  <parameter type="int" name="valueID" />
  <parameter type="float" name="value" />
</message>
<message name ="SetDouble" id="14" error="-9999">
  <returntype type="int" />
  <parameter type="int" name="valueID" />
  <parameter type="double" name="value" />
</message>
<message name ="SetString" id="15" error="error">
  <returntype type="string" />
  <parameter type="int" name="valueID" />
  <parameter type="string" name="value" />
</message>

<message name ="SetByteArray" id="16" error="255">
  <returntype type="int" />
  <parameter type="int" name="valueID" />
  <parameter type="byte[]" name="value" />
```

```
</message>
<message name ="SetIntArray" id="17" error="-9999">
  <returntype type="int" />
  <parameter type="int" name="valueID" />
  <parameter type="int[]" name="value" />
</message>
<message name ="SetFloatArray" id="18" error="-9999">
  <returntype type="int" />
  <parameter type="int" name="valueID" />
  <parameter type="float[]" name="value" />
</message>
<message name ="SetDoubleArray" id="19" error="-9999">
  <returntype type="int" />
  <parameter type="int" name="valueID" />
  <parameter type="double[]" name="value" />
</message>
<message name ="SetStringArray" id="20" error="error">
  <returntype type="string" />
  <parameter type="int" name="valueID" />
  <parameter type="string[]" name="value" />
</message>

<message name ="SetByteVoid" id="21">
  <returntype type="void" />
  <parameter type="int" name="valueID" />
  <parameter type="byte" name="value" />
</message>
<message name ="SetIntVoid" id="22">
  <returntype type="void" />
  <parameter type="int" name="valueID" />
  <parameter type="int" name="value" />
</message>
<message name ="SetFloatVoid" id="23">
  <returntype type="void" />
  <parameter type="int" name="valueID" />
  <parameter type="float" name="value" />
</message>
<message name ="SetDoubleVoid" id="24">
  <returntype type="void" />
  <parameter type="int" name="valueID" />
  <parameter type="double" name="value" />
</message>
<message name ="SetStringVoid" id="25">
  <returntype type="void" />
  <parameter type="int" name="valueID" />
  <parameter type="string" name="value" />
</message>

<message name ="SetByteArrayVoid" id="26">
```

```
    <returntype type="void" />
    <parameter type="int" name="valueID" />
    <parameter type="byte[]" name="value" />
  </message>
  <message name ="SetIntArrayVoid" id="27">
    <returntype type="void" />
    <parameter type="int" name="valueID" />
    <parameter type="int[]" name="value" />
  </message>
  <message name ="SetFloatArrayVoid" id="28">
    <returntype type="void" />
    <parameter type="int" name="valueID" />
    <parameter type="float[]" name="value" />
  </message>
  <message name ="SetDoubleArrayVoid" id="29">
    <returntype type="void" />
    <parameter type="int" name="valueID" />
    <parameter type="double[]" name="value" />
  </message>
  <message name ="SetStringArrayVoid" id="30">
    <returntype type="void" />
    <parameter type="int" name="valueID" />
    <parameter type="string[]" name="value" />
  </message>
</messagetable>
```

Listing A.3: MessageTest.cs

```
using System;
using System.Collections.Generic;
using System.Text;
using System.IO;
using odonet_cs;
using NUnit.Framework;
namespace odonet_cs_test
{
        [TestFixture]
        public class MessageTest
        {
                private Message message;
                [SetUp]
                public void Setup()
                {
                        this.message = new Message();
                }
                /// <summary>
                /// Tests everything which the constructors of the different message types
                /// have in common.
                /// </summary>
```

```
[Test]
public void BaseConstructorTest()
{
        MemoryStream data =
                (MemoryStream)TestHelper.getField(message, "data");
        BinaryReader reader =
                (BinaryReader)TestHelper.getField(message, "reader");
        BinaryWriter writer =
                (BinaryWriter)TestHelper.getField(message, "writer");
        Assert.IsNotNull(data);
        Assert.IsNotNull(reader);
        Assert.IsNotNull(writer);
        //length and id field have been reserved and message type was written
        //12 = 3 * sizeof(int)
        Assert.AreEqual(12, data.Position);
        //check that length and id are initially set to 0
        data.Position = data.Seek(0, SeekOrigin.Begin);
        Assert.AreEqual(0, message.ReadInt());
        //8 = 2 * sizeof(int)
        data.Position = data.Seek(8, SeekOrigin.Begin);
        Assert.AreEqual(0, message.ReadInt());
}
/// <summary>
/// Additionally tests if the type was set correctly.
/// </summary>
[Test]
public void OrdinaryMessageConstructorTest()
{
        MessageType type = (MessageType)TestHelper.getField(message, "type");
        Assert.AreEqual(MessageType.DEFAULT, type);
}
/// <summary>
/// Additionally tests if the type was set correctly.
/// </summary>
[Test]
public void ReturnMessageConstructorTest()
{
        Message dummyMessage = new Message();
        dummyMessage.SequenceNumber = 666;
        message = new ReturnMessage(dummyMessage);
        MessageType type = (MessageType)TestHelper.getField(message, "type");
        int sequenceNumber =
                (int)TestHelper.getField(message, "sequenceNumber");
        Assert.AreEqual(MessageType.RETURN, type);
        Assert.AreEqual(666, sequenceNumber);
}
[Test]
public void SetReadPointerToBeginningOfPayloadTest()
{
```

```
            message.WriteInt(12);
            message.WriteInt(0);
            message.WriteInt(Int32.MaxValue);
            message.WriteInt(22);
            message.WriteInt(924324);
            message.WriteInt(-12443434);
            MemoryStream data = (MemoryStream)TestHelper.getField(message, "data");
            message.SetReadPointerToBeginningOfPayload();
            //12 = 3 * sizeof(int)
            Assert.AreEqual(12, data.Position);
    }
    [Test]
    public void WriteAndReadByteTest()
    {
            message.WriteByte(112);
            message.SetReadPointerToBeginningOfPayload();
            Assert.AreEqual(112, message.ReadByte());
    }
    [Test]
    public void WriteAndReadIntTest()
    {
            message.WriteInt(12);
            message.SetReadPointerToBeginningOfPayload();
            Assert.AreEqual(12, message.ReadInt());
    }
    [Test]
    public void WriteAndReadFloatTest()
    {
            message.WriteFloat(3.14152F);
            message.SetReadPointerToBeginningOfPayload();
            Assert.AreEqual(3.14152F, message.ReadFloat());
    }
    [Test]
    public void WriteAndReadDoubleTest()
    {
            message.WriteDouble(3.14152D);
            message.SetReadPointerToBeginningOfPayload();
            Assert.AreEqual(3.14152D, message.ReadDouble());
    }
    [Test]
    public void WriteAndReadStringTest()
    {
            message.WriteString("chanter␣alouette");
            message.SetReadPointerToBeginningOfPayload();
            Assert.AreEqual("chanter␣alouette", message.ReadString());
    }
    [Test]
    public void WriteAndReadByteArrayTest()
    {
```

```
        byte[] bytes = new byte[] { 0, 112, 110, 255 };
        message.WriteByteArray(bytes);
        message.SetReadPointerToBeginningOfPayload();
        Assert.AreEqual(bytes, message.ReadByteArray());
}
[Test]
public void WriteAndReadIntArrayTest()
{
        int[] ints = new int[] { 0, 1, 2, Int32.MaxValue, Int32.MinValue };
        message.WriteIntArray(ints);
        message.SetReadPointerToBeginningOfPayload();
        Assert.AreEqual(ints, message.ReadIntArray());
}
[Test]
public void WriteAndReadFloatArrayTest()
{
        float[] floats = new float[] {
                (float)Math.PI, (float)Math.E, 0F, 3.14F
        };
        message.WriteFloatArray(floats);
        message.SetReadPointerToBeginningOfPayload();
        Assert.AreEqual(floats, message.ReadFloatArray());
}
[Test]
public void WriteAndReadDoubleArrayTest()
{
        double[] doubles = new double[] {
                Math.PI, Math.E, 0D, 3.14D, Double.MaxValue,
                Double.MinValue, Double.PositiveInfinity
        };
        message.WriteDoubleArray(doubles);
        message.SetReadPointerToBeginningOfPayload();
        Assert.AreEqual(doubles, message.ReadDoubleArray());
}
[Test]
public void WriteAndReadStringArrayTest()
{
        string[] strings = new string[] {
                "wenn␣du␣denkst", "du␣denkst",
                "dann␣denkst␣du␣nur", "du␣denkst"
        };
        message.WriteStringArray(strings);
        message.SetReadPointerToBeginningOfPayload();
        Assert.AreEqual(strings, message.ReadStringArray());
}
[Test]
public void WriteAndReadMultipleValuesTest()
{
        byte[] bytes = new byte[] { 0, 112, 110, 255 };
```

```
            float[] floats = new float[] {
                    (float)Math.PI, (float)Math.E, 0F, 3.14F
            };
            double[] doubles = new double[] {
                    Math.PI, Math.E, 0D, 3.14D, Double.MaxValue,
                    Double.MinValue, Double.PositiveInfinity
            };
            message.WriteByteArray(bytes);
            message.WriteInt(12);
            message.WriteFloat(0F);
            message.WriteInt(Int32.MaxValue);
            message.WriteFloatArray(floats);
            message.WriteDoubleArray(doubles);
            message.SetReadPointerToBeginningOfPayload();
            Assert.AreEqual(bytes, message.ReadByteArray());
            Assert.AreEqual(12, message.ReadInt());
            Assert.AreEqual(0F, message.ReadFloat());
            Assert.AreEqual(Int32.MaxValue, message.ReadInt());
            Assert.AreEqual(floats, message.ReadFloatArray());
            Assert.AreEqual(doubles, message.ReadDoubleArray());
    }
    [Test]
    public void PackAndUnpackMessageTest()
    {
            message.WriteInt(12314);
            message.WriteInt(33432);
            message.WriteInt(88);
            message.WriteInt(44);
            message.WriteInt(255);
            byte[] messageAsByteArray = message.Serialize();
            Message unpackedMessage = Message.Deserialize(messageAsByteArray);
            MemoryStream messageData =
                    (MemoryStream)TestHelper.getField(message, "data");
            MemoryStream unpackedMessageData =
                    (MemoryStream)TestHelper.getField(unpackedMessage, "data");
            Assert.AreEqual(
                    messageData.GetBuffer(),unpackedMessageData.GetBuffer()
            );
            Assert.AreEqual(MessageType.DEFAULT, unpackedMessage.Type);
            /*
             * not neccessary, buffer equality is already tested above
             *
            unpackedMessage.SetReadPointerToBeginningOfPayload();
            Assert.AreEqual(12314, unpackedMessage.ReadInt());
            Assert.AreEqual(33432, unpackedMessage.ReadInt());
            Assert.AreEqual(88, unpackedMessage.ReadInt());
            Assert.AreEqual(44, unpackedMessage.ReadInt());
            Assert.AreEqual(255, unpackedMessage.ReadInt());
             * */
```

```
                }
                [Test]
                public void PackAndUnpackReturnMessageTest()
                {
                        ReturnMessage returnMessage = new ReturnMessage(message);
                        byte[] messageAsByteArray = returnMessage.Serialize();
                        Message unpackedMessage = Message.Deserialize(messageAsByteArray);
                        Assert.AreEqual(MessageType.RETURN, unpackedMessage.Type);
                }
        }
}
```

Listing A.4: LocalSimpleStubTest.cs

```
using System;
using System.Collections;
using System.Collections.Generic;

using NUnit.Framework;
using odonet_cs;
using Lore.Util;
using Simple;

namespace odonet_cs_test
{
        //Shortened Version of the testclass for systemtests via loopback interface.
        //In the full version, all parameter and return types are tested.
        [TestFixture]
        public class LocalSimpleStubTest
        {
                private SimpleClientStub clientStub;
                private SimpleServerStub serverStub;
                private SimpleStubTestHelper clientStubTestHelper;
                private SimpleStubTestHelper serverStubTestHelper;
                //These delegates must be stored, because the timeout test needs different
                //callbacks than SetAndGetIntTest. If NUnit provided a TestSetup-attribute
                //as Zanebug does, that solution would be better.
                private SimpleClientStub.SetIntEventDelegate setIntEventDelegate;
                private SimpleClientStub.SetIntEventDelegate timeoutEventDelegate;
                public LocalSimpleStubTest()
                {
                        Logger.AddListener(
                                new LogListenerConsole(Logger.Priority.Warning, true)
                        );
                }
                /// <summary>
                /// Creates server and client and registers the server-side delegates used
                /// in the other tests.
                /// </summary>
```

```
[TestFixtureSetUp]
public void TestFixtureSetUp()
{
        this.clientStubTestHelper = new SimpleStubTestHelper();
        this.serverStubTestHelper = new SimpleStubTestHelper();
        //load config from file
        Configuration configuration =
                new Configuration("../../connection.conf");
        //get all objects of type "ClientStub" from config file
        IList<ConfigObject> allClientStubConfigObjects =
                configuration.Select("network", "ClientStub", "*");
        IList<ConfigObject> allServerStubConfigObjects =
                configuration.Select("network", "ServerStub", "*");
        NetworkConfigEntry clientStubConfig = new NetworkConfigEntry();
        NetworkConfigEntry serverStubConfig = new NetworkConfigEntry();
        //select the first ClientStub and the first ServerStub and save
        //its attributes
        allClientStubConfigObjects[0].Deserialize(clientStubConfig);
        allServerStubConfigObjects[0].Deserialize(serverStubConfig);
        this.clientStub = new SimpleClientStub(clientStubConfig);
        this.serverStub = new SimpleServerStub(serverStubConfig);
        #region register server-side delegates
        this.setIntEventDelegate =
                new SimpleClientStub.SetIntEventDelegate(
                        clientStubTestHelper.clientStub_SetIntEvent
                );
        clientStub.SetStringArrayEvent +=
                new SimpleClientStub.SetStringArrayEventDelegate(
                        clientStubTestHelper.clientStub_SetStringArrayEvent
                );
        clientStub.GetIntEvent +=
                new SimpleClientStub.GetIntEventDelegate(
                        clientStubTestHelper.clientStub_GetIntEvent
                );
        clientStub.GetStringArrayEvent +=
                new SimpleClientStub.GetStringArrayEventDelegate(
                        clientStubTestHelper.clientStub_GetStringArrayEvent
                );
        #endregion
        bool clientStubStarted = clientStub.Start();
        bool serverStubStarted = serverStub.Start();
}
/// <summary>
/// Realeases allocated resources by calling dispose on server- and clientStub.
/// </summary>
[TestFixtureTearDown]
public void TestFixtureTearDown()
{
        serverStub.Dispose();
```

```
        clientStub.Dispose();
}
/// <summary>
/// Creates 1 server and CLIENTCOUNT clients. After the clients connected,
/// firstly all clients will be disposed and after that the server will be
/// disposed. No exception shall occur.
/// </summary>
[Test]
public void DisposeClientsFirstTest()
{
        const int CLIENTCOUNT = 100;
        ArrayList clients = new ArrayList(CLIENTCOUNT);
        NetworkConfigEntry clientStubConfig = new NetworkConfigEntry();
        clientStubConfig.type = TRANSPORTTYPES.TcpServer;
        clientStubConfig.port = 4321;
        NetworkConfigEntry serverStubConfig = new NetworkConfigEntry();
        serverStubConfig.type = TRANSPORTTYPES.TcpClient;
        serverStubConfig.servernames = new string[] { "localhost" };
        serverStubConfig.port = 4321;
        serverStubConfig.timeout = 25;
        SimpleClientStub simpleClientStub =
                new SimpleClientStub(clientStubConfig);
        SimpleServerStub simpleServerStub;
        try
        {
                for (int i = 0; i < CLIENTCOUNT; i++)
                {
                        simpleServerStub =
                                new SimpleServerStub(serverStubConfig);
                        clients.Add(simpleServerStub);
                }

                for (int i = 0; i < CLIENTCOUNT; i++)
                {
                        simpleServerStub = (SimpleServerStub)clients[i];
                        simpleServerStub.Dispose();
                }

                simpleClientStub.Dispose();
        }
        catch (Exception)
        {
                //test not passed
                Assert.Fail();
        }
}
/// <summary>
/// Creates 1 server and 100 clients. After the clients connected,
/// firstly the server will be disposed and after that all clients will be
```

```
/// disposed. No exception shall occur.
/// </summary>
[Test]
public void DisposeServerFirstTest()
{
        const int CLIENTCOUNT = 100;
        ArrayList clients = new ArrayList(CLIENTCOUNT);
        NetworkConfigEntry clientStubConfig = new NetworkConfigEntry();
        clientStubConfig.type = TRANSPORTTYPES.TcpServer;
        clientStubConfig.port = 4321;
        NetworkConfigEntry serverStubConfig = new NetworkConfigEntry();
        serverStubConfig.type = TRANSPORTTYPES.TcpClient;
        serverStubConfig.servernames = new string[] { "localhost" };
        serverStubConfig.port = 4321;
        serverStubConfig.timeout = 25;
        SimpleClientStub simpleClientStub =
                new SimpleClientStub(clientStubConfig);
        SimpleServerStub simpleServerStub;
        try
        {
                for (int i = 0; i < CLIENTCOUNT; i++)
                {
                        simpleServerStub =
                                new SimpleServerStub(serverStubConfig);
                        clients.Add(simpleServerStub);
                }

                simpleClientStub.Dispose();

                for (int i = 0; i < CLIENTCOUNT; i++)
                {
                        simpleServerStub = (SimpleServerStub)clients[i];
                        simpleServerStub.Dispose();
                }
        }
        catch (Exception)
        {
                //test not passed
                Assert.Fail();
        }
}
/// <summary>
/// This test is abusing the SetInt(...) method for testing timeout behavior,
/// because the signature fits best. The first parameter is an integer,
/// which will be returned unmodified by the server-side delegate.
/// The second parameter is an integer which will tell the server thread
/// how long it shall sleep.
///
/// The test first assures, that the test setup works correctly by sending
```

```
        /// a message, telling the server not to sleep. After that, a message is sent
        /// to the server, telling it to sleep for 500 ms (which is the default timeout
        /// value). The result must be the value defined in SimpleInterface.xml
        /// (-9999), signaling a timeout.
        /// </summary>
        [Test]
        public void TimeoutTest()
        {
                this.timeoutEventDelegate =
                        new SimpleClientStub.SetIntEventDelegate(
                                clientStubTestHelper.clientStub_TimeoutEvent
                        );
                this.clientStub.SetIntEvent -= this.setIntEventDelegate;
                this.clientStub.SetIntEvent += this.timeoutEventDelegate;
                int expected = 1;
                int result = serverStub.SetInt(expected, 0);
                Assert.AreEqual(expected, result);
                result = serverStub.SetInt(expected, 500);
                Assert.AreEqual(-9999, result);
        }
        [Test]
        public void SetAndGetIntTest()
        {
                this.clientStub.SetIntEvent -= this.timeoutEventDelegate;
                this.clientStub.SetIntEvent += this.setIntEventDelegate;
                int valueID = serverStubTestHelper.CreateRandomInt();
                int expected = serverStubTestHelper.CreateRandomInt();
                Assert.AreEqual(0, serverStub.SetInt(valueID, expected));
                int result = serverStub.GetInt(valueID);
                Assert.AreEqual(expected, result);
        }
        [Test]
        public void SetAndGetStringArrayTest()
        {
                int valueID = serverStubTestHelper.CreateRandomInt();
                string[] expected = serverStubTestHelper.CreateRandomStringArray();
                Assert.AreEqual("OK", serverStub.SetStringArray(valueID, expected));
                string[] result = serverStub.GetStringArray(valueID);
                Assert.AreEqual(expected, result);
        }
    }
}
```

B Tabellen

Art der Zusicherung	Methoden	Bemerkung
Wertgleichheit (Equality)	AreEqual, AreNotEqual	zahlreiche Überladungen für den Vergleich von primitiven Werten oder Objekten
Referenzgleichheit (Identity)	AreSame, AreNotSame, Contains	Contains überprüft ob eine Liste ein best. Objekt enthält.
Vergleich (Comparison)	Greater, Less	Vergleich von primitiven Werte oder Objekten, die IComparable implementieren
Typ (Type)	InstanceOf, NotInstanceOf, ...	
Bedingung (Condition)	IsTrue, IsFalse, IsNull, ...	
Hilfsmethoden (Utility)	Fail, Ignore	Fail lässt einen Test fehlschlagen und kann dazu benutzt werden, eigene Assertion-Methoden zu schreiben. Mit Ignore kann ein Test zur Laufzeit ignoriert werden.

Tabelle B.1: NUnit-Assertions (Auszug)

[]

Attribut	Funktion
TestFixture	Markiert eine Testklasse als solche.
Test	Markiert eine Testmethode als solche.
SetUp	Markiert eine Methode, die vor jedem Test aufgerufen werden soll; pro TestFixture nur einmal erlaubt!
TearDown	Markiert eine Methode, die nach jedem Test aufgerufen werden soll; pro TestFixture nur einmal erlaubt!
TestFixtureSetUp	Markiert eine Methode, die vor dem ersten Test einer Testklasse aufgerufen werden soll; pro TestFixture nur einmal erlaubt!
TestFixtureTearDown	Markiert eine Methode, die nach dem letzten Test einer Testklasse aufgerufen werden soll; pro TestFixture nur einmal erlaubt!
ExpectedException	Markiert eine Methode, bei deren Ausführung eine Ausnahme (Exception) erwartet wird. Damit kann überprüft werden, ob Ausnahmen korrekt ausgelöst und propagiert werden.
Ignore	Deaktiviert einen Test.

Tabelle B.2: NUnit-Attribute (Auszug)

Danksagung

Mein Dank gilt, neben meinem Betreuer Benjamin, der immer Zeit für ausgiebige Diskussionen und hilfreiche Ratschläge fand, auch besonders Stefan Spickereit für die Unterstützung bei der Entwicklung der XSLT-Stylesheets und der Implementierung der Netzwerkbibliothek in C++. Weiterhin möchte ich mich bei meiner Freundin für ihre Geduld und ihr Verständniss bedanken.

Erklärungen zum Urheberrecht

Es wurden keine Texte oder Bilder übernommen.